Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events

Dissertation

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

von
Michael Eckert

22. Oktober 2008

Erstgutachter:	Prof. Dr. François Bry (Ludwig-Maximilians-Universtät München)
Zweitgutachter:	Prof. Dr. Martin L. Kersten (Centrum Wiskunde & Informatica, Amsterdam)
Tag der mündlichen Prüfung:	09. Dezember 2008

Im Gedenken und Liebe/
In memory and love

Käthe Tomin, geb. Strathmann
∗ 15.06.1913 † 27.07.2005

Abstract

The emergence of event-driven architectures, automation of business processes, drastic cost-reductions in sensor technology, and a growing need to monitor IT systems (as well as other systems) due to legal, contractual, or operational considerations lead to an increasing generation of events. This development is accompanied by a growing demand for managing and processing events in an automated and systematic way. *Complex Event Processing (CEP)* encompasses the (automatable) tasks involved in making sense of all events in a system by deriving higher-level knowledge from lower-level events *while the events occur*, i.e., in a timely, online fashion and permanently.

At the core of CEP are queries which monitor streams of "simple" events for so-called *complex events*, that is, events or situations that manifest themselves in certain combinations of several events occurring (or not occurring) over time and that cannot be detected from looking only at single events. Querying events is fundamentally different from traditional querying and reasoning with database or Web data, since event queries are standing queries that are evaluated permanently over time against incoming streams of event data. In order to express complex events that are of interest to a particular application or user in a convenient, concise, cost-effective and maintainable manner, special purpose *Event Query Languages (EQLs)* are needed.

This thesis investigates practical and theoretical issues related to querying complex events, covering the spectrum from language design over declarative semantics to operational semantics for incremental query evaluation. Its central topic is the development of the high-level event query language XChange^{EQ}.

In contrast to previous data stream and event query languages, XChange^{EQ}'s language design recognizes the four querying dimensions of data extractions, event composition, temporal relationships, and, for non-monotonic queries involving negation or aggregation, event accumulation. XChange^{EQ} deals with complex structured data in event messages, thus addressing the need to query events communicated in XML formats over the Web. It supports deductive rules as an abstraction and reasoning mechanism for events. To achieve a full coverage of the four querying dimensions, it builds upon a separation of concerns of the four querying dimensions, which makes it easy-to-use and highly expressive.

A recurrent theme in the formal foundations of XChange^{EQ} is that, despite the fundamental differences between traditional database queries and event queries, many well-known results from databases and logic programming are, with some importance changes, applicable to event queries. Declarative semantics for XChange^{EQ} are given as a (Tarski-style) model theory with accompanying fixpoint theory. This approach accounts well for (1) data in events and (2) deductive rules defining new events from existing ones, two aspects often neglected in previous work of semantics of EQLs.

For the evaluation of event queries, this work introduces operational semantics based on an extended and tailored form of relational algebra and query plans with materialization points. Materialization points account for storing and maintaining information about those received events that are relevant for, i.e., can contribute to, future query answers, as well as for an incremental evaluation that avoids recomputing certain intermediate results. Efficient state maintenance in incremental evaluation is approached by "differentiating" algebra expressions, i.e., by deriving expressions for computing only the changes to materialization points. Knowing *how long* an event is relevant is a prerequisite for performing garbage collection during event query evaluation and also of central importance for developing cost-based query planners. To this end, this thesis introduces a notion of *relevance* of events (to a given query plan) and develops methods for determining *temporal relevance*, a particularly useful form based on time-related information.

Zusammenfassung

Die Einführung von ereignisgesteuerten Architekturen, die Automatisierung von Geschäftsprozessen, kostengünstige Sensortechnik und die rechtlich, vertraglich oder betrieblich bedingte Überwachung von Informationssystemen erzeugen mehr und mehr Ereignisse. Diese Entwicklung wird begleitet von einer zunehmenden Notwendigkeit, Ereignisse systematisch und automatisch zu verwalten und zu verarbeiten. *Complex Event Processing (CEP)* hat zur Aufgabe höheres, wertvollen Wissen aus Ereignissen abzuleiten *während diese passieren*, also kontinuierlich und zeitnah.

Zentral in CEP sind Anfragen, die Ströme von „einfachen" Ereignissen überwachen, um sogenannte komplexe Ereignisse (engl.: complex events) zu erkennen. Komplexe Ereignisse sind Ereignisse oder Situationen, die sich durch das gemeinsame, zeitlich verteilte Auftreten (oder Nicht-Auftreten) von mehreren Ereignissen äußern und nicht erkannt werden können, indem man nur einzelne Ereignisse betrachtet. Anfragetechniken für Ereignisse unterscheiden sich grundlegend von traditionellen Anfrage- und Schlußtechniken für Datenbanken oder Web-Daten, denn Ereignisanfragen sind stehende Anfragen, die mit Zeit fortwährend gegen einen ankommenden Strom von Ereignisdaten ausgewertet werden. Zur bequemen, kostengünstigen und leicht wartbaren Beschreibung von komplexen Ereignissen, die für eine bestimmte Anwendung oder einen bestimmten Benutzer von Interesse sind, bedarf es spezieller *Ereignisanfragesprachen*.

Die vorliegende Arbeit beschäftigt sich mit praktischen und theoretischen Fragestellungen zu Anfragen nach komplexen Ereignissen. Das abgedeckte Themenspektrum reicht von Sprachdesign über deklarative Semantik bis zu operationaler Semantik für eine inkrementelle Anfrageauswertung. Der rote Faden der Arbeit ist die Entwicklung der höheren Ereignisanfragesprache XChange$^{\text{EQ}}$.

Im Gegensatz zu vorherigen Datenstrom- und Ereignisanfragesprachen trägt das Sprachdesign von XChange$^{\text{EQ}}$ den vier Anfragedimensionen Rechnung: Extraktion von Daten, Komposition von Ereignissen, zeitliche Zusammenhänge und, für nicht-monotone Anfragen mit Negation oder Aggregation, Akkumulation von Ereignissen. XChange$^{\text{EQ}}$ kann mit komplex strukturierten Daten in Ereignissen umgehen, wie sie häufig in Ereignissen, die in XML-Formaten über das Web kommuniziert werden, zu finden sind. Als Abstraktions- und Schlußmechanismus werden deduktive Regeln unterstützt. Um eine vollständige Abdeckung der vier Anfragedimensionen zu erreichen, baut XChange$^{\text{EQ}}$ auf einer Trennung dieser Dimensionen auf, was die Sprache leicht benutzbar und ausdrucksstark macht.

Ein Leitmotiv in den formalen Grundlagen von XChange$^{\text{EQ}}$ ist, daß trotz der grundlegenden Unterschiede zwischen traditionellen Datenbankanfragen und Ereignisanfragen viele bekannte Ergebnisse aus der Forschung über Datenbanken und Logikprogrammierung —mit einigen wichtigen Änderungen— auf Ereignisanfragen anwendbar sind. Die deklarative Semantik von XChange$^{\text{EQ}}$ wird als (Tarski-)Modelltheorie mit begleitender Fixpunkttheorie angegeben. Dieser Ansatz eignet sich besonders zur Behandlung von (1) Daten in Ereignissen und (2) deduktive Regeln, die neue Ereignisse aus existierenden ableiten. Diese beiden Aspekte wurden in vorherigen Arbeiten zur Semantik von Ereignisanfragesprachen oft vernachlässigt.

Zur Auswertung von Ereignisanfragen führt diese Arbeit eine operationale Semantik ein, die auf einer erweiterten und spezialisierten Form von relationaler Algebra sowie auf Anfrageplänen mit ausgezeichneten Punkten für Materialisierung aufbaut. Die Materialisierungspunkte dienen dazu, Informationen über Ereignisse, die relevant für zukünftige Antworten sein können, zu speichern. Ferner sind sie zweckdienlich für eine inkrementelle Auswertung, die eine wiederholte Berechnung bestimmter Zwischenergebnisse vermeidet. Die effiziente Aktualisierung der Zustände von Materialisierungspunkten basiert auf der „Differenzierung" von Algebraausdrücke, d.h., darauf neue Ausdrücke abzuleiten, die nur die nötigen Änderungen berechnen. Eine Speicherbereinigung während der Anfrageauswertung setzt voraus zu wissen, wie lange ein Ereignis relevant ist. Dieses Wissen ist auch von zentraler Bedeutung, um kostenbasierte Anfrageoptimierer zu entwickeln. Dazu führt diese Arbeit einen Begriff der *Relevanz* von Ereignissen (bezüglich einem gegebenen Anfrageplan) ein und entwickelt eine Methode zur Bestimmung der *temporalen Relevanz*, einer besonders nützlichen Form, die auf zeitbezogenen Informationen basiert.

Acknowledgments

The work presented in this thesis would not be what is today without the support and contribution of many people. Foremost, I would like to express my gratitude to my thesis advisor *François Bry* for his continuous support and encouragement. The many fruitful discussion with him have shaped all aspects of this work as well as my life in research and teaching. I am also indebted to *Martin Kersten* for taking on the burden of being the external reviewer for this thesis.

It has been a great pleasure to work with all the people in the programming and modeling languages research group at the University of Munich (LMU). *Sacha Berger*, *Norbert Eisinger*, *Tim Furche*, *Alex Kohn*, *Jakub Kotowski*, *Benedikt Linse*, *Bernhard Lorenz*, *Hans Jürgen Ohlbach*, *Paula-Lavinia Pătrânjan*, *Edgar-Philipp Stoffel*, *Klara Weiand*, and *Christoph Wieser*, have been fellow researchers and together created an excellent work environment. I greatly enjoyed the time spent talking to you, listening to you, and teaching with you. The administrative and technical staff of the group, *Stefanie Heidmann*, *Martin Josko*, *Ellen Lilge*, *Uta Schwertel*, and *Ingeborg von Troschke*, have contributed greatly to making things run smoothly in our group. I thank them for this and for sometimes even managing to make filling out the most confusing and boring forms a fun experience.

The following students have worked with me on their theses and deserve my thanks: *Hendrik Grallert* has worked on the demonstration of an XChange use case, *Stephan Leutenmayr* has surveyed languages for Web Service composition, *Fatih Coşkun* has developed declarative semantics for the pattern-based updates of XChange, and *Hai-Lam Bui* is currently working on a comparison of event query languages in practical applications.

The following colleagues have worked with me on joint publications and on the project proposal "Management of Events on the Web (MEOW)" and not been mentioned yet: *Bruno Berstel*, *Philippe Bonnard*, *Thomas Geyer*, *Martin Kluge*, *Peter Knaack*, *Inna Romanenko*, *Nikolaus Seifert*, and *Sanja Vranes*. I thank them as well as the numerous *colleagues from the REWERSE Network of Excellence*.

The research in this work has been partly funded by the *European Commission* and the *Swiss Federal Office for Education and Science* within the 6th Framework Programme project REWERSE (number 506779, see `http://rewerse.net`). Not only has the funding of this project made many travels to conferences possible; the network of colleagues that has grown in this project has greatly helped improved this work. I am also indebted to the *Studienstiftung des Deutschen Volkes* and the *German-American Fulbright Commission* for their financial and non-financial support during my studies prior to the work on this thesis.

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

Contents

I Complex Event Processing 15

1 Introduction 17
- 1.1 Applications involving Complex Events . . . 19
- 1.2 Event Query Languages . . . 23
- 1.3 Motivation . . . 25
- 1.4 Contributions . . . 27
- 1.5 Organization of this Thesis . . . 28

2 From Data to Events on the Web 31
- 2.1 Data on the Web . . . 32
- 2.2 Reasoning on the Web . . . 34
- 2.3 Reactivity on the Web . . . 40
- 2.4 Events on the Web . . . 41

3 State of the Art 47
- 3.1 Querying Complex Events Unraveled . . . 47
- 3.2 Composition-Operator-Based Event Query Languages . . . 49
- 3.3 Data Stream Query Languages . . . 56
- 3.4 Production Rule Languages . . . 63
- 3.5 Comparison . . . 68
- 3.6 Hybrid Approaches . . . 71

4 Background: Xcerpt and XChange 75
- 4.1 Xcerpt: Querying and Reasoning on the Web . . . 75
- 4.2 XChange: Reactivity on the Web . . . 80
- 4.3 Summary . . . 85

II XChange^{EQ}: An Expressive High-Level Event Query Language 87

5 Language Design 89
- 5.1 Four Dimensions of Querying Events . . . 89
- 5.2 Separation of Concerns: Expressivity and Ease-of-Use . . . 90
- 5.3 Seamless Integration into the (Reactive) Web . . . 91
- 5.4 Reasoning with Events . . . 92
- 5.5 Declarative Language and Simplicity . . . 92
- 5.6 Semantics . . . 93
- 5.7 Extensibility . . . 93

6 Syntax and Informal Semantics of XChange^{EQ} 95
6.1 Representation of Events . . . 95
6.2 Querying Simple Events . . . 96
6.3 Absolute Timer Events . . . 97
6.4 Deductive Rules for Events . . . 100
6.5 Reactive Rules for Events . . . 101
6.6 Composition of Events . . . 103
6.7 Relative Timer Events . . . 104
6.8 Temporal (and other) Relationships . . . 106
6.9 Event Accumulation . . . 109
6.10 Stratification: Limits on Recursion . . . 110
6.11 Rel^{EQ}: A Simplified, Relational Variant . . . 111

7 Use Cases 113
7.1 Business Activity Monitoring in Order Processing . . . 113
7.2 Monitoring of Sensor Events in a SCADA Application . . . 118

III Declarative Semantics 123

8 Declarative Semantics: Motivation and Overview 125
8.1 Motivation . . . 125
8.2 Requirements and Desiderata . . . 126
8.3 Overview of our Approach . . . 127

9 Model Theory 129
9.1 Basic Definitions: Time and Events . . . 129
9.2 Matching and Constructing Simple Events . . . 130
9.3 Interpretation and Entailment . . . 132
9.4 Models . . . 133

10 Fixpoint Theory 139
10.1 Stratification: Limits on Recursion . . . 139
10.2 Immediate Consequence Operator . . . 141
10.3 Fixpoint Interpretation . . . 142

11 Theorems 143
11.1 Well-Defined and Unambiguous Semantics . . . 143
11.2 Suitability for Event Streams . . . 143
11.3 Proof of Theorem 1 . . . 144
11.4 Proof of Theorem 2 . . . 147

IV Incremental Evaluation of Complex Event Queries 149

12 Operational Semantics: Requirements and Overview 151
12.1 Basics of Event Query Evaluation . . . 151
12.2 Desiderata and Design Decisions . . . 155
12.3 General Ideas . . . 158

13 Complex Event Relational Algebra (CERA) 161
13.1 Expressing Event Queries in Relational Algebra . . . 161
13.2 Formal Definition of CERA . . . 167
13.3 Temporal Preservation in CERA . . . 174
13.4 Translation of single XChange^{EQ} rules into CERA . . . 175

14 Query Plans and Incremental Evaluation **185**
14.1 Incremental Evaluation Explained . . . 185
14.2 Query Plans with Materialization Points . . . 186
14.3 Incremental Evaluation and Finite Differencing . . . 189
14.4 Translation of Rule Programs into Query Plans . . . 193
14.5 Query Plan Rewriting . . . 197
14.6 Relationship with other Approaches . . . 200

15 Relevance of Events **203**
15.1 Motivation: Garbage Collection, Query Planning . . . 203
15.2 Temporal Relevance: Problem Definition . . . 204
15.3 Determining Temporal Relevance Conditions . . . 207
15.4 Algorithm . . . 211
15.5 Using Temporal Relevance for Garbage Collection . . . 216
15.6 Outlook: Variations and other Forms of Relevance . . . 217
15.7 Related Work . . . 220

16 Proof-of-Concept Implementation **223**
16.1 Using the XChange^{EQ} Prototype . . . 223
16.2 Building XChange^{EQ} from the Source Code . . . 227
16.3 Overview of Source Code . . . 228
16.4 Current Limitations of Prototype . . . 233

V Conclusions and Outlook **237**

17 Language Design Revisited **239**
17.1 Separation of Concerns w.r.t. Four Dimensions . . . 239
17.2 Event Data and Querying Events in XML formats . . . 241
17.3 Event Composition and Garbage Collection . . . 243
17.4 Temporal Relationships . . . 244
17.5 Event Accumulation . . . 244
17.6 Support for Deductive and Reactive Rules . . . 245
17.7 Formal Semantics . . . 246
17.8 Extensibility . . . 247
17.9 Summary . . . 247

18 Future Work on XChange^{EQ} **249**
18.1 Rules and Language Design . . . 249
18.2 Time . . . 251
18.3 Data and State . . . 253
18.4 Event Query Evaluation . . . 255

19 Research Perspectives in Complex Event Processing **261**
19.1 Querying Complex Events . . . 261
19.2 Event Query Evaluation and Optimization . . . 264
19.3 Beyond Querying of Complex Events . . . 268
19.4 Complex Event Processing an a Larger Context . . . 269

20 Summary and Conclusion **273**

VI Appendix **275**

A EBNF Grammars **277**

A.1 Conventions on EBNF Notation 277

A.2 (Core) Xcerpt Term Grammar 277

A.3 XChange$^{\mathrm{EQ}}$Grammar 278

A.4 Rel$^{\mathrm{EQ}}$ Grammar 279

B Proofs about Operational Semantics **281**

B.1 Temporal Preservation of CERA 281

B.2 Correspondence between Relations and Σ, τ 281

Bibliography **283**

About the Author **301**

Part I

Complex Event Processing

Chapter 1

Introduction

Events are omnipresent in modern computer and information systems and play a key role in driving their behavior. Current and future systems face an increasing generation of events, which can be attributed to the following factors:

- a shift from the traditional user-request-driven interaction on the Web towards a more dynamic, event-driven interaction caused by Web 2.0 technology and subscription-oriented services such as RSS feeds [VH07],
- a demand for automation of business processes [Hav05], which is accompanied by an adoption of Web Service standards [ACKM04] and Service Oriented Architecture (SOA) [PvdH07],
- emerging popularity of Event-Driven Architecture (EDA), especially in the context of distributed systems [MFP06], Enterprise Application Integration (EAI) [HW03], and as a complement to Service Oriented Architecture (SOA) [Sch03],
- drastic reductions in the cost of sensor hardware and new sensor technologies such as Radio Frequency Identification (RFID) [NMMK07] and intelligent networked sensor nodes [ASSC02], which lead to a wide-spread deployment of sensors,
- a need to monitor IT systems (as well as other systems) due to legal, contractual, or operational considerations, often in near real-time [Luc02, McC02].

Unsurprisingly, this increase in the amount of generated events is accompanied by a growing demand for managing and processing events in an automated and systematic way. The (automatable) tasks involved in making sense of all events in a system by deriving higher-level knowledge from lower-level events *while the events occur*, i.e., in a timely, online fashion and permanently, are commonly summarized under the term *Complex Event Processing (CEP)*.

In particular, CEP involves monitoring streams of "simple" (or atomic) events for *complex (or composite) events*, that is, events or situations that cannot be detected from looking only at single events. They manifest themselves in certain combinations of several events occurring (or not occurring) over time and have to be inferred. Combinations of events that are of interest to a particular application or user are commonly expressed in a special purpose language, a so-called (Complex) *Event Query Language (EQL)*.

It is worth emphasizing that event queries are standing queries that are evaluated in an online manner while events occur. This online processing distinguishes event queries and CEP from other technologies like queries in databases or data warehouses. While these also sometimes work with event-related data (e.g., a customer's history of purchases), their processing is done in an off-line manner after the events have occurred and they conceptually work with "spontaneous," one-time queries rather than standing queries. This difference between traditional database queries and event queries is also illustrated in Figure 1.1, and will be discussed in more detail later on (Section 1.2.2).

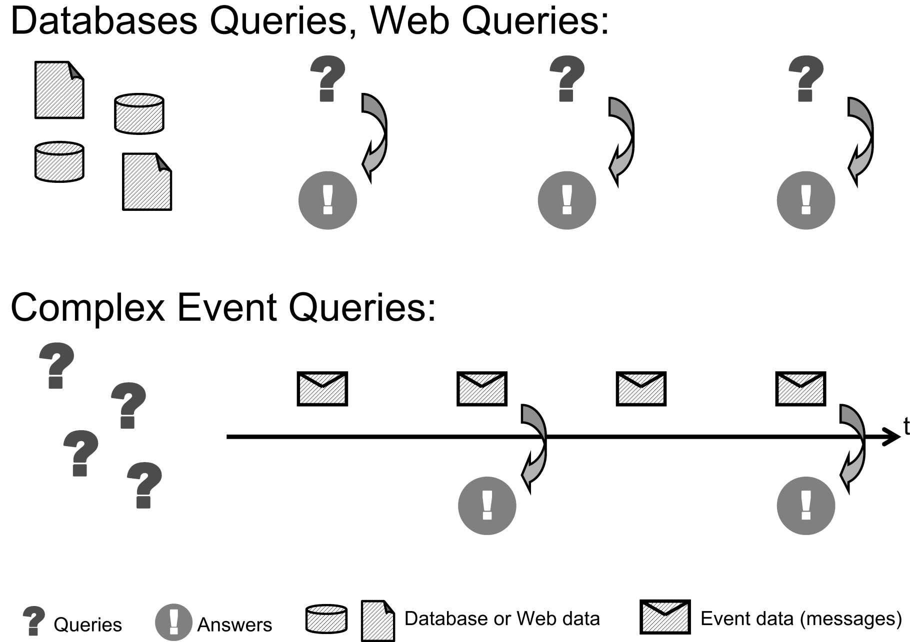


Figure 1.1: Illustration of the difference between traditional database queries and event queries.

This thesis investigates practical and theoretical issues related to querying complex events, covering the spectrum from query language design over declarative semantics to techniques for query evaluation (operational semantics). Its central topic is the development of a high-level event query language called XChange^{EQ}. Outstanding features of its language design are that it deals with complex structured data in event messages, thus addressing the need to query events communicated in XML formats over the Web, that it supports deductive rules as an abstraction and reasoning mechanism for events, and that it builds upon a separation of concerns in the supported query features, which makes it easy-to-use and highly expressive. XChange^{EQ} builds on the Web query language Xcerpt for accessing data in events; it design and foundations are however generic in the sense that other query languages could be used for accessing event data, too. In particular, XChange^{EQ} is not tied tightly to the XML data model, and extensions of Xcerpt to support querying RDF [BFB$^+$05, BFLP08] can be carried over into XChange^{EQ} without much difficulty and effort.

A recurrent theme in the formal foundations of XChange^{EQ} is that, despite the fundamental differences between traditional database queries and event queries, many well-known results from traditional query answering from databases and logic programming are, with some importance changes, applicable to event queries. Declarative semantics for XChange^{EQ} are given as a (Tarski-style) model theory with accompanying fixpoint theory. This approach accounts well for (1) data in events and (2) deductive rules defining new events from existing ones, two aspects often neglected in previous work of semantics of event query languages.

Operational semantics are based on an extended and tailored form of relational algebra called CERA and query plans that consist of equations in the style of (materialized) views. Building on the foundation of database queries is not just helpful for understandability, it also lets event queries benefit from many results in database research (e.g., join algorithms, adaptive query evaluation). Further, the uniformity in the foundations of event queries and traditional queries is beneficial in systems and languages where event and non-event data is processed together — and queries should be optimized or analyzed together. This is quite common, especially for Event Condition Action (ECA) rules, where the E-part is an event query, the C-part a traditional query, and the parts share information through variable bindings.

Since complex event queries are standing queries that are evaluated over time against a stream of incoming events and that produce answers (complex events) on the fly and promptly, their evaluation is very different from traditional queries. It requires storing and maintaining information about those received events that are relevant for, i.e., can contribute to, future query answers. For efficiency reasons, it usually also has to store and maintain intermediate results. To this end, we introduce so-called materialization points in query plans that account for the information that must be stored and maintaining. The state maintenance of these materialization points is approached by "differentiating" relational algebra expressions (more precisely, CERA expressions), i.e., by deriving expressions for computing the state changes.

In this setting of evaluation over time, knowing *how long* an event is relevant (to a given set of queries) is a prerequisite for performing garbage collection during event query evaluation and also of central importance for developing cost-based query planners. To this end, this thesis introduces a notion of *relevance* of events (to a given set of complex event queries) and develops methods for determining *temporal relevance*, a particularly useful form based on time-related information. A prototype implementation of XChange$^{\text{EQ}}$ accompanies this thesis and demonstrates feasibility of the proposed language and operational semantics.

1.1 Applications involving Complex Events

Event-driven applications are characterized by reacting automatically to events, which may be generated by the application software itself or come from external sources such as other communicating applications, human users, or sensors. Complex Events in an event-driven application usually originate from a need to acquire information that is spread over several simple events (as well as other data sources).

We detail now some classes of applications that heavily involve complex events and thus can benefit strongly from using dedicated Complex Event Processing technology such as an event query language. We also mention some of the typical challenges these applications pose.

1.1.1 Asynchronous Messaging in Heterogeneous Environments

Different applications often use messages or events as a means to communicate and exchange information. This is particularly common in Enterprise Application Integration (EAI) scenarios [HW03]. For example it is common that applications such as a Web application for online shopping, a desktop application for call-center agents entering purchase orders, an order processing system, and several inventory management applications must exchange information. Asynchronous messaging offers a flexible and scalable way to realize this integration. Asynchronous messaging also has performance advantages over other ways when the applications are distributed (i.e., run on different servers)

However, the different applications that are to be integrated often have been developed and evolved independently leading to a heterogeneous environment. To integrate them properly, event messages must be transformed, enriched, and combined on their way from one application to another. Three common CEP-related problems and solutions encountered in such scenarios that can benefit from using an event query language are:

Event Translation Due to heterogeneity, applications often speak a different "language," i.e., use different data formats and schemas for their event messages. When one application should react to events generated by another application, a translation of the events can be required as an intermediate step. For example, an order processing application might produce an order event in XML containing separate elements for a customer's first name, last name, street, city, and zip code. A shipping application that reacts to such an order event might however expect data in a different format, e.g., just a single element with the whole address. Translation is not limited to schemas and can also concern data formats, e.g., one application expecting events as XML data, another as Java objects. Strictly speaking, such an event translation is not yet Complex

Event Processing. Rather it is “Simple” Event Processing since the information in an event is not combined with other information. However, such “simple” transformations, which would typically be realized using an XML query and transformation language such as XSLT, XQuery, or Xcerpt, should of course be supported by CEP technology, and we will revisit this issue in Section 1.3 of this chapter and also in Chapter 5.

Enrichment of Event Content with External Information In the case of a translation, events contain all necessary information and it is just structured differently. Sometimes events produced by one application simply lack information expected by a consuming application. In this case it is necessary to “enrich” the event with information from external sources. For example, a Web shop application might generate an order event containing only a customer number, while the order processing system expects a customer’s address. A query to a customer database must be used to obtain a customer’s address based on her number and provide the event with the missing information. Importantly, the enrichment of the event happens outside this individual applications, because changing the applications might be expensive or impossible.

Event Composition Lastly, there are situations where several events must be combined into one single event. Consider a shopping application where products from several groups (e.g., books, CDs, DVDs) can be purchased together and are delivered together; there is a single Web shop and a single shipping system, but each product group might have a separate order processing system. Order events from the Web shop need to first be split for processing individual items in their appropriate order processing system. The output events from the order processing systems then in turn must be pieced back together into a single (composed) shipping event for the shipping system. Note that there are two reasons why event composition is needed: first there is a lack of information and the difference from event enriching is that the required information comes from other events and not a database or other static source; second there is a need to synchronize, i.e., execution of the common task shipping needs to be blocked until all order processing systems have completed their tasks.

Enterprise Application Integration (EAI) based on asynchronous messaging is a broad subject and there are many other CEP-related issues. We refer to [HW03] for a deeper introduction; the three issues above correspond to its patterns Message Translator, Content Enricher, and Aggregator, respectively.

An important challenges for CEP in EAI settings is that event data typically has a complex, often document-like, structure. In particular, event messages are often in an XML format such as SOAP[1] [G$^+$03], Common Base Event (CBE) [IBM04], or Electronic Business using XML (ebXML) [ebX]. In addition to the complex structure, there is also great heterogeneity with respect to schemas used in different applications, data formats, and communication platforms. A further issue is that, in a business context, CEP must pay attention to issues such as reliability (no lost events even in the presence of communication failures), security (encryption and authentication of events), and transactions (atomicity of operations, order of events, compensation of uncompleted actions etc.).

1.1.2 Monitoring in Computing Environments

In the EAI examples above, events have been drivers of primary (or operative) business functions (such a selling goods). Event processing functionality (such as translation, aggregation) has been used in realizing a specific business process. To properly manage their operations and detect existing or emerging problems and opportunities, however, companies also require monitoring of their business processes.

When performed in (near) real-time, this monitoring is called Business Activity Monitoring (BAM) and driven by events [McC02, GDP$^+$06, JP06]. Single events are usually not very mean-

[1]SOAP originally was an acronym for “Simple Object Access Protocol.” However, this meaning has been dropped with version 1.2, since it was considered misleading.

ingful in a BAM context; instead many events have to be summarized to yield so-called key performance indicators (KPIs). An example of a KPI could be the average time taken between a customer's order and shipping, and a manager might want to receive a warning automatically and immediately when this number exceeds a limit of 24 hours. Since we aggregate (here: compute an average), events from many business processes (and possibly also other sources such as IT infrastructure) need to be combined and this is where CEP and event queries are useful.

BAM is just one particular instance of event-based monitoring in computing environments. Another example is in the context of Service Level Agreements (SLAs) [PS06]: with an SLA, the provider of a service (e.g., a credit card verification service) gives (contractual) guarantees to service users concerning aspects such as the service's availability (e.g., uptime of 99.9%), or performance (e.g., average response time of less than 10 seconds). User will often want to monitor if a provider keeps the agreed service level and this again requires processing of events and associated information (e.g., service requests, replies, and the time elapsed between them).

It is worth noting in this context that one application's or system's complex event can be conceived as an (incoming) atomic event by another application or system. For example an SLA monitoring application may work hard to derive an SLA violation from many single events; once it is detect it can be fed however as an atomic event into, e.g., a BAM application.

Important for monitoring applications such as BAM or SLA monitoring is that complex event queries are easy to change and adapt because they have a strong dependency on the monitored entities. For example, a change in a business process will usually lead to corresponding changes in BAM queries in order to keep them working as intended. Further, defining "meaningful" queries, i.e., that are useful to business, can be challenging. In the case of BAM, meaningful KPIs need to be identified (usually together with thresholds) from a business perspective and then translated into event queries. In the case of SLAs, translating an agreed service level (e.g., uptime of 99.9%) into event queries raises difficult questions of how to actually measure it.

1.1.3 Processing of Sensor Data

Events from the physical world are observed in computers through sensors connected to communication networks. Sensor data is susceptible to faulty and imprecise measurements due to physical limits (e.g., limited precision of a temperature sensor), outside influence (e.g., dust triggering a smoke alarm), malfunctions (e.g., caused by a blown fuse or other hardware defects), as well as lost data (e.g., due to communication failures) [JAF+06, EN03]. Further, sensor data provides typically rather "low-level" input that is used to detect "higher-level" application-specific situations. For example, a burglar alarm system wants to detect break-ins (high-level situation) and is provided with low-level data such as (intensity of) vibrations at doors and windows. The task of combining data form multiple sensors is often also called Multi-Sensor Data Fusion.

Concrete examples of applications involving sensor networks are: supply chain management using sensors to obtain location information of products equipped with RFID tags [NMMK07], tracking and monitoring of cargo containers [SM06], intelligent intrusion detection on sensor-equipped fences [WTV+07], and monitoring of industrial facilities such as factories or power plants with Supervisory Control and Data Acquisition systems (see next section).

In contrast to the applications previously discussed, sensor data is mainly non-symbolic data (i.e., measured numerical data) and can require rather application-specific computations. Examples of application-specific computations include smoothing of a series of sensor readings or elimination of outliers in measurements.

A new aspect is also introduced by intelligent sensors (also known as motes or sensors nodes), which combine an embedded processor, wireless networking facilities, and sensors, and are usually powered by battery [ASSC02]. Power consumption characteristics of these sensors give rise to a number of optimization problems. Computation is far less expensive than communication leading to a trade-off between local computation at the sensor and computation at a central node. Short-range communication can be less expensive than long-range communication, which becomes relevant when event detection requires data from several sensors that are in proximity (the central node is usually further away than close-by sensor nodes) [WTV+07].

1.1.4 Supervisory Control and Data Acquisition

Supervisory Control and Data Acquisition (SCADA) [BW03] Systems are event-driven systems used in monitoring and controlling large and distributed industrial installation and infrastructures such as factories, manufacturing lines, power plants, oil and gas pipelines, and facilities in transport systems (e.g., airports, train stations). They collect and interpret data such as meter readings from sensors and equipment status reports from embedded control units with the aim of controlling the overall system. It must be emphasized that they work on a supervisory level: A SCADA system in a train station will, for example, monitor an elevator for its status but not control its normal operations (taking requests, moving between floors, opening and closing doors). However in the case of a fire emergency, the SCADA system will intervene or override the normal elevator control. This may entail ensuring that the elevator is parked on a floor with a safe evacuation route for passengers, opening doors, and switching the elevator off to avoid further use.

A trend in SCADA systems is to strive for generic systems [DS99a] and more recently to base these systems on Web-standards such as XML and Web Services [BLO$^+$08]. Instead of developing a SCADA system from scratch for some given individual facility, a generic system is customized to meet the needs of the facility is used. Currently customization is mainly based on "plugging-in" modules written in some general purpose programming language (typically the same language used to implement the generic part of the SCADA system). Complex event query languages, together with reactive rules, are expected to play an important role for customizing generic SCADA systems in the near future, since they are more flexible and easier to develop and maintain than procedural code.

CEP in SCADA systems primarily addresses the need to derive higher-level, symbolic events (e.g., fire) from lower-level, numeric sensory input (e.g., temperature, smoke), which has already been mentioned for other applications involving sensor data. Higher-level symbolic events are in particular important for programming automatic or semi-automatic reactions ranging from simple rules (e.g., upon a fire alarm activate sprinkler) to complex workflows (e.g., emergency plans that involve evacuation, shutting of fire doors, notification of rescue personal, etc.). Often inference can be context-sensitive and depend on circumstances such as different modes of operation in a system (e.g., summer vs. winter operations, normal vs. emergency operations).

Monitoring and control in SCADA systems often involves human operators. They thus also raise issues concerning the presentation and visualization of information derived from events. Human operators must be enabled to quickly judge situations and make decisions.

SCADA systems also face issues of heterogeneity. Manufacturers of different devices often use different communications protocols. Recently, this issue is approached by building on open, Web-based standards for data formats and communication protocols. Often are employed to translate to and from existing proprietary protocols. An example of this is the Facility Control Markup Language (FCML) [BLO$^+$08], which provides an XML format for events and HTTP-based communication.

Since SCADA systems are often used for detecting and reacting to emergencies, a further unusual issue is found in them. Some CEP-related functionality is vital to the purpose of the system, e.g., detection of emergencies, but is rarely exercised or used. Defects, be they from design flaws in the software or from malfunctioning hardware, are thus easily overseen. Because field tests of such emergencies can be potentially expensive, the use high-level languages and formalism is particularly desirable to find defects through simulation and verification beforehand.

1.1.5 Summary: Causes of Complex Events

Not every event-driven application needs complex event processing. For example, almost all current frameworks for programming graphical user interface are event-driven, but usually there are no complex events involved. One can argue that the notion of what constitutes an event is often a design choice made during application development. Accordingly, it is possible in many cases to choose a design where simple events satisfy the needs of their consumers and no need to combine information from several simple events, i.e., generate complex events, arises. However we have

just seen some cases where such a design is not possible or desirable. Looking at these example, the need for complex events can typically be attributed to at least one of the following:

- **Fusion:** In sensor networks, information is spread over many events by inherent limitations such as geographical distribution (one sensor can only measure at one location) or physical and technical limits (e.g., measurement errors). These inherent limitations make a fusion of event data through complex event processing necessary.
- **Integration:** In enterprise application integration, complex events are not inherent: applications could have been designed or be changed so that they can be integrated only using simple events and no complex events. However this is often not possible (one cannot forsee all later integration needs in the development process), not cost-efficient (changing applications is expensive), or desirable (a design using only simple events is not necessarily better because the granularity may be too coarse or too fine). Complex event processing is needed to integrate applications and systems, especially when their integration has not been anticipated in their original design.
- **Monitoring:** When events are used for monitoring, this often entails that they are used for purposes other then intended. For example, business events have the primary purpose of driving a business process. Business Activity Monitoring is not their primary purpose, only a secondary use of them. Through this secondary use comes again the problem that information is spread over several events and thus a need for complex event processing. This is in contrast to the previous causes of fusion and integration, where complex event processing was needed to realize primary functionality.

Note that the boundaries of the three causes are often overlapping and depend on the perspective on the system.

1.2 Event Query Languages

Event Query Languages are a vital part of Complex Event Processing, allowing users to specify known queries or patterns of events.

1.2.1 Benefits of Event Query Languages

When an application requires detection of a given set of complex events, it is of course possible to program their detection "manually" in a general purpose programming language (usually the language used to program the application itself). There are however compelling reasons to use a dedicated high-level event query language, such as the one developed in this thesis:

- **Ease of programming:** An event query language allows to program on a high abstraction level that focuses on the query's logic rather than programming on a low level an actual detection algorithm. Even when they do not aim for high performance, detection algorithms are usually complicated since they involve state maintenance (storing of events and partial answers) and require a form of manual memory management (removing events and partial answers that have become irrelevant).
- **Flexibility and Maintainability:** High-level languages make the resulting code more flexible and easier to maintain. This is especially important in dynamic environments and organizations where software has to be adapted frequently to meet changing requirements or has to be integrated with other (independently developed) software. In particular, use of an event query language leads to a decoupling of the event logic from the rest of the application logic. Event logic (i.e., event queries) can be exchanged independently of application logic, possibly even at runtime without recompiling as well as stopping and restarting the whole application.

- **Optimization:** An event query language gives rise to query compilers and evaluation engines that do automatic performance optimization, thus taking this burden off the programmer's shoulders. This is especially important since many query optimization techniques conflict with maintainability when they are programmed manually. A particularly severe case of this is multi-query optimization, which exploits similarities between several queries. Since it leads to a sharing of data structures and operations between queries, a change in a single query will potentially affect all other queries.

- **Program (or Query) Analysis:** The restricted nature of special purpose event query languages makes reasoning about (some) interesting properties of event queries more feasible than in general purpose programming languages. Examples of interesting query analysis tasks for event queries include the identification of temporal bounds on the relevance of event as introduced in Chapter 15, which is a foundation for automatic garbage collection and cost-based planning, or the verification of correctness with respect to some specification as in [EPBS07].

1.2.2 Relationship to Traditional (Database) Query Languages

Querying events has much in common with traditional database query languages for relational data (e.g., SQL [GUW01], datalog [AHV95]), Web data (e.g., XQuery [B$^+$07b], Xcerpt [SB04, Sch04]), and Semantic Web data (e.g, SparQL [PS08], OWL-QL [FHH04], Xcerpt [BFLP08, DW07]). In particular, event messages are, as seen in Section 1.1, often in a conventional data format such as XML. Processing of these messages includes tasks addressed by traditional query languages such as selecting, transforming, and aggregating event data.

However, there are important discrepancies between the capabilities and premises of traditional query languages and the specific requirements in querying events:

- Events are received over time in a stream-like manner, while in a database all facts are available at once and usually stored on disk.

- Event streams are unbounded into the future, potentially infinite, whereas databases are finite. This has especially consequences for non-monotonic query features such as negation or aggregation. It also entails a need for garbage collection of events that become irrelevant over time.

- Relationships between events such as temporal order or causality play an important role for querying events. In databases, relationships between facts are usually part of the data (e.g., references with foreign keys).

- Timing of answers has to be considered when querying events: event queries are evaluated continously against the event stream and generate answers at different times. These answers may trigger actions such as updates to a database. Typically actions are sensitive to ordering; hence it is important *when* an answer is detected.

- Query evaluation and optimization for event streams require different methods than for databases. In event streams a large number of (standing) queries are evaluated against small pieces of incoming data (events). Evaluation is thus usually data-driven rather than query-driven. Evaluation also involves state maintenance for a (partial) history of events and to avoid recomputing intermediate results. Many optimizations rely on exploiting similarities between queries rather than clustering and indexing data.

These discrepancies make traditional query languages and engines unsuitable for the task of querying events and entail a need for a tailored event query language and engine.

1.3 Motivation

Development of XChange^{EQ} is guided by the motivation of providing a high-level language for querying events. It aims at a language design that is easy to use and allows query programmers to work on a high abstraction level, as well as at strong formal foundations. Previous work on event query languages, which is surveyed in Chapter 3, has paid considerably less attention to language design and little work has been done on formal foundations such as declarative semantics or mathematically clean operational semantics.

Our experience with other event query languages shows that many complex event queries are hard to express, not expressible at all, or prone to misinterpretations (cf. Chapters 3 and 17). This also includes our own, earlier approach for querying complex events in the reactive language XChange [Păt05, BEP06b] based on composition operators, which is described in [Eck05, BEP06a, BEP06b]. XChange^{EQ}'s language design incorporates lessons learned from these difficulties and uses logic-like formulas (in a tailored syntax) to express complex event queries rather than a multitude of algebra-like composition operators. XChange^{EQ} integrates as a "sub-language" into the reactive language XChange, replacing its original event composition operators [Eck05, BEP06a, BEP06b]. This relationship is also reflected in the name of XChange^{EQ}, where EQ stands for *E*vent *Q*ueries.

XChange^{EQ} goes beyond the state of the art in event query languages in the following ways:

Pattern-Based Querying of Event Data Events, or more precisely their representations, contain data that describes the context and circumstances of the event. A purchase order[2] event, for example, will contain data describing buyer, seller, product, quantity, agreed price, and so on. As we have seen in the application examples in the previous section, this data is often provided in an XML format and can thus have a fairly complex, often document-like structure. XChange^{EQ} addresses the need for dealing with semi-structured data by embedding the Web query language Xcerpt [SB04, Sch04] for specifying classes of relevant events, extracting data (in form of variable bindings), and constructing new events. Due to its pattern-based approach, where queries work like a form or template that is put on top of the queried data, Xcerpt is easy to use and has a very intuitive visualization, visXcerpt [BBS03, BBSW03]. Xcerpt also aims at being versatile in the Web data formats that can be queries; extensions of Xcerpt to query, e.g., RDF data [BFLP08] are therefore also immediately applicable to XChange^{EQ}.

Although it seems obvious that data in events must be accessed as part of (complex) event queries, most early event query languages have neglected this aspect. Later event query languages usually consider data in an event as a collection of attribute-value pairs or as a fixed arity tuple, which is a considerable step forward but does not address the specifics of query semi-structured data such as XML.[3]

Although XChange^{EQ} embeds Xcerpt and endorses its pattern-based approach, its design and its foundations are generic in the sense that it can in principle also embed other query languages for accessing and constructing event data. The common denominator is that the embedded query language must expose in some form bindings for variables.

High Expressivity in all Querying Dimensions As part of work on XChange^{EQ}, we have identified the following four complementary dimensions (or aspects) of event queries: data extraction, event composition, temporal (and other) relationships between events, and event accumula-

[2] A purchase order (PO) is formal document used in commerce constituting a buyer's legal offer to a supplier to buy products or services (with specified prices, quantities, etc.). In eCommerce, such documents are often written in a standardized XML format in order to be machine processable.

[3] An exception are the original event composition operators in XChange, which use Xcerpt to query XML data in same fashion as XChange^{EQ}. Note that despite its title, [BKK04] does not address querying events that include XML data; rather it is about events such as insertions or mouse clicks that are generated by interacting with an XML document according to the Document Object Model [Pix08]. Work in [BFF$^+$07] adds construct for temporal windows on streams to XQuery, but the stream there is a *single* XML document (with XML tags being "events"), not a stream of events where each event is a separate XML document.

tion. How well an event query language covers each dimension gives a practical measure for its expressiveness. The four dimensions are explained in detail in Chapter 5.

XChange^{EQ}'s language design enforces a separation of the four querying dimensions. This yields in a very clear language design, syntax and semantics that are easy to read and understand, and gives programmers the benefit of a separation of concerns. Event more importantly, this separation, where each dimension of a query is independent and arbitrary combinations are possible, contributes to XChange^{EQ}'s high expressivity. Deficiencies in the expressivity as well as possible misinterpretations found in other event query languages can often be attributed to the fact that the querying dimensions are mixed (cf. Chapter 17). Furthermore, the separation contributes to XChange^{EQ}'s extensibility, which is useful, e.g., for embedding a temporal reasoner such as CaTTS [BRS05] that can support application-specific calendric notions such as "business day" or "lecture period" or for supporting non-temporal relationships between events such as causality.

Deductive Rules XChange^{EQ} supports deductive rules for defining new, "virtual" events from the existing ones (i.e., those that are received in the incoming event streams), much in the same fashion one uses views (or rules) in databases to define new, derived data from existing base data. Support for deductive rules in an event query language is highly desirable: Rules serve as an abstraction mechanism, making query programs more readable. They allow to define higher-level application events from lower-level (e.g., database or network) events. Different rules can provide different perspectives (e.g., of end-user, system administrator, corporate management) on the same (event-driven) system. Rules allow to mediate between different schemas for event data. Additionally, rules can be beneficial when reasoning about (vertical) causal relationships of events [Luc02].

Only very few event languages support purely deductive rules, and programmers often have to resort to using reactive rules [BBB$^+$07] to the same effect. We argue however, that deductive (event) rules are inherently different from reactive rules because they aim at expressing "virtual events," not actions. Accordingly and importantly, deductive rules are free of side-effects. Implementing deductive rules using reactive rules blurs this distinction with negative consequences for development, maintainability, and optimization. Furthermore, deductive rules can be given very clear logic semantics, while reactive rules only have execution semantics, which are intrinsically more complicated.

Seamless Integration with Reactive Rule Language Deductive rules can be used to derive new events, but they cannot specify to take certain actions such as updating a database in response to events. Despite the advantages of deductive rules, event-based systems therefore usually still require reactive rules, typically Event-Condition-Action (ECA) rules, or some similar formalism.[4] To this end, XChange^{EQ} integrates seamlessly as a "sub-language" into XChange [Pǎt05, BEP06b].[5]

XChange is a reactive rule language addressing the need for both local (at a single Web node) and global (distributed over several Web nodes) evolution and reactivity on the Web. It is based on ECA rules of the form "`ON` *event query* `IF` *Web query* `DO` *action*." When events answering the event query are received and the Web query is successful (i.e., has a non-empty result), the rule's action is executed.

XChange, like XChange^{EQ}, builds on the pattern-based approach of Xcerpt for querying data, and additionally provides for pattern-based updating of Web data [Pǎt05, Coş07]. Development of

[4] An example of a similar, alternative formalism is implicit invocation or "callback (registration)," where a component can "register" a procedure for an event, and generation of the event by other components causes this procedure to be called (see, e.g., [GS94]). The difference between reactive rules and implicit invocation is that implicit invocation has to be programmed manually (requiring use of pointers or references to procedures or objects implementing a specific event listener interface) and, as a consequence, hides the association of events and actions (procedures) relatively deep in the application code.

[5] As mentioned previously, XChange^{EQ} replaces XChange's original event composition operators, which are described in [Eck05, BEP06a, BEP06b]. These original composition operators (together with other approaches based on composition operators) will be discussed in Chapter 3, and also compared with XChange^{EQ} in Chapter 17.

Xcerpt, XChange, and now XChange^{EQ} follows the vision of a stack of homogenous languages for performing common tasks on Web data such as querying, transforming, and updating static data, as well as reacting to changes, propagating updates, and querying events. When a programmer has mastered the basics of querying Web data with Xcerpt's query terms, she can progress quickly and with smooth transitions to more advanced tasks. XChange as well as Xcerpt are covered in Chapter 4.

Further to XChange^{EQ}'s role as a sub-language in XChange, it can also be used as a stand-alone event query engine or in other ECA languages or frameworks such as the General Semantic Web ECA Framework described in [MAA05a, MAA05b] and its later incarnations MARS [BFMS06] and r3 [AA07].

Formal Foundations Traditional (non-event) query languages have very strong formal foundations. While a number of event query languages have been proposed both from research and industry, the field of event querying still lacks comparable formal foundations. This lack of formal foundations has also been a topic discussed on a recent Dagstuhl seminar on event processing [CEvA07]. Most notably, both declarative and operational semantics are desirable.

This thesis attempts to rectify this situation by providing both declarative semantics and operational semantics for XChange^{EQ}. It shows that many well-known approaches and results from traditional database queries apply, or apply with changes, to event queries, too. In doing so, it also shows where new concepts and methods are needed (e.g., data-driven evaluation, event relevance) — and where existing ones can be leveraged (e.g., model-theoretic semantics, join algorithms, program and query transformations).

While the formal foundations are developed for XChange^{EQ}, both the foundations themselves and results obtains from them are important in their own right and transfer also to other event query languages and evaluation formalisms. The background of XChange^{EQ} as a concrete language makes the discussions easier to understand and helps to indicate the practical relevance of these investigations. Last not least, XChange^{EQ} as a high-level language raises some issues that are not present in lower-level and less expressive event query languages such as rule chaining or relevance of events.

1.4 Contributions

The previous section has already given a first glimpse into the contributions of this thesis, mainly from the focus of language development. We now elaborate the contributions in detail, and with a more technical focus.

Survey and Comparison of Event Query Languages A number of event query languages have been developed in the past, but so far there are no comprehensive surveys and comparisons of these languages. We therefore survey existing languages and identify three prevalent "styles" of languages: composition operators, data stream languages, and production rules. XChange^{EQ}, the language developed in this work, introduces a fourth style, where queries are written in way that is reminiscent of logical formulas (but in a tailored and more human-friendly syntax). We also compare these different styles analytically in terms language design, expressiveness, semantics, and the environments in which they are used.

Four Dimensions of Querying Events As part of the language design of XChange^{EQ}, we identify four dimensions that a sufficiently expressive event query language must cover. These dimensions are called data extraction, event composition, temporal and other relationships between events, and event accumulation. At the very heart of XChange^{EQ} is the idea that a language must separate these four dimensions in order to achieve full expressivity. This separation of dimensions is in contrast to other, previous event query languages.

Event Query Language XChange^{EQ} The central topic of this work is the development of the event query language XChange^{EQ}. The distinctive features of XChange^{EQ} have already been discussed in Section 1.3: pattern-based querying of event data in particular in XML formats, high expressivity, support for deductive rules, seamless integration with the reactive Web language XChange, and strong formal foundations. We introduce the syntax and informal semantics of XChange^{EQ} in a tutorial-like manner. Use cases illustrate the use of XChange^{EQ} in practical applications and serve to substantiate its claims with regards to expressivity and ease-of-use.

Declarative Semantics We specify declarative semantics for XChange^{EQ} by a (Tarski-style) model theory with accompanying fixpoint theory. This approach has the important advantage that is accounts well for data in events and deductive rules, two aspects that have often been neglected in semantics of other event query languages. While the model-theoretic approach is a well-established for traditional, non-event query and rule languages, its application to an event query language is novel and we highlight the extensions that are necessary. We also prove that our declarative semantics are suitable for querying events that arrive over time in unbounded event streams.

Complex Event Relational Algebra As first corner stone of operational semantics, we introduce a variant of relational algebra called CERA. The core idea is to obtain an algebra that is expressive enough to evaluate XChange^{EQ} but still restricted enough to be suitable for the incremental, step-wise evaluation that is required for complex event queries. We also provide details on how XChange^{EQ} rules are translated into CERA expression and prove correctness with respect to the declarative semantics.

Query Plans and Incremental Evaluation The second corner stone of our operational semantics are query plans with so-called materialization points and their incremental evaluation. Incremental evaluation depends heavily on which intermediate results we "materialize," that is, store across the different evaluation steps. The materialization points of our query plans serve to capture this information and also address other issues such as chaining of deductive rules and multi-query optimizations. Our approach with materialization points is thus more flexible than related approaches for event query evaluation because it is not bound to a fixed strategy for which intermediate results are materialized. The changes that must be made in each evaluation step to the contents of a materialization point are described by algebra expressions that are obtained through a technique called finite differencing.

Temporal Relevance for Garbage Collection The third corner stone of our operational semantics is to enable garbage collection based on the relevance of events and intermediate results. In other event query languages, the issue of garbage collection is usually not considered because these languages either are less expressive (so that temporal relevance is trivial) or because they require more work from programmers (e.g., explicit specification of time windows or manual garbage collection). We develop a precise definition of relevance and temporal relevance and develop a method for statically (i.e., at compile time) determining temporal relevance based temporal conditions in queries.

1.5 Organization of this Thesis

This thesis is structured into five parts, each consisting of several chapters.

The first part (*Complex Event Processing*), which includes this chapter, introduces into the topic of CEP. Chapter 2 sets the stage for CEP on the Web and describes some basic Web technology that is relevant in the scope of this thesis. Chapter 3 surveys and compares existing event query languages. Chapter 4 describes the Web query language Xcerpt and the reactive Web language XChange as needed for understanding XChange^{EQ}.

The second part (*XChange^{EQ}: An Expressive High-Level Event Query Language*) develops the language XChange$^{\text{EQ}}$. Chapter 5 discusses the language design of XChange$^{\text{EQ}}$. Chapter 6 presents the syntax and informal semantics of XChange$^{\text{EQ}}$. Chapter 7 illustrates the use of XChange$^{\text{EQ}}$ with practical use cases.

The third part (*Declarative Semantics*) develops declarative semantics for XChange$^{\text{EQ}}$. Chapter 8 motivates the need for declarative semantics, establishes requirements and desiderata, and gives an overview of our approach. Chapter 9 defines the model theory, which is the heart of our declarative semantics. Chapter 10 defines a fixpoint theory (based on the model theory), which serves to obtain a single model for stratified XChange$^{\text{EQ}}$ programs. Chapter 11 shows that this model is well-defined and unambiguous, and that our declarative semantics is suitable for queries against event streams.

The fourth part (*Operational Semantics*) develops operational semantics that are the basis of an incremental evaluation of XChange$^{\text{EQ}}$. Chapter 12 establishes requirements and gives an overview of the used approach. Chapter 13 defines a variant of relational algebra called CERA, shows that CERA is suitable for an incremental evaluation of event queries, and describes the translation of XChange$^{\text{EQ}}$ rules into CERA expressions. Chapter 14 introduces the notion of query plans with materialization points and their incremental evaluation. Chapter 15 defines the notion of relevance as needed for garbage collection and develops an algorithm for statically determining temporal relevance. Chapter 16 describes the proof-of-concept implementation of XChange$^{\text{EQ}}$ that accompanies this thesis.

The fifth part (*Conclusions*) rounds off this thesis. Chapter 17 revisits language design and illustrates the advantages of XChange$^{\text{EQ}}$ over other event query languages. Chapter 18 discusses opportunities for future work on XChange$^{\text{EQ}}$. Chapter 19 broadens the scope and discusses more general research perspectives in Complex Event Processing. Chapter 20 is a summary and conclusion.

Chapter 2

From Data to Events on the Web

The World Wide Web (WWW or Web, for short) [BL99] started out as a distributed hypertext system. User would retrieve documents as well as other information, e.g., pictures, from (Web) servers on the Internet using the Hypertext Transfer Protocol (HTTP) [F+99]. Documents and other files on the Web were identified by Uniform Resource Locators (URLs) [BLMM94], which have been subsequently replaced by Uniform Resource Identifiers (URIs) [BLFM05] and Internationalized Resource Identifier (IRIs) [DS99b]. Documents were written in the Hypertext Markup Language (HTML) [RHJ99], which provided means for structuring text for its visual presentation. Important to the Web's success, HTML provided the ability to link from one document to any other document anywhere on the Web using its URL. Users could also interact with documents, e.g., enter keywords for searching document collections or submit orders in e-commerce catalogs. To this end, HTML allowed forms in documents, HTTP provided means to send information to Web servers, and Web servers would generate documents dynamically using, e.g., programs supporting the Common Gateway Interface (CGI).

While this is still a common use of the Web today, it has become much more. Today, its is an infrastructure of any kind of information system, ranging from systems that are accessible world-wide (e.g., Web sites for electronic shopping) over cross-enterprise systems (e.g., for sharing demand information between a manufacturer and its suppliers) to private systems (e.g., intranets in an enterprise or home networks).

Such information systems do much more than just providing access to data and documents. Much data in these systems is dynamic and constantly changing (e.g., the status of a customer's order) and dependencies between data require other data to be created, deleted, or changed (e.g., completion of an order leads to a new invoice). Further, information systems provide services to interact with data (e.g., submit a new order) and, possibly also the outside world (e.g., ship ordered item by mail).

Such a active, dynamic Web information system is full of events and all these events require proper processing. The necessary processing includes, for example, generation of events, communication of events, logging of events, reacting to events, and of course detection of complex events.

In this chapter, we summarize the foundations of the Web as relevant for this thesis. We follow a path from representing data on the Web [ABS00] (Section 2.1) over reasoning with Web data [BBFS05, FLB+06, BEE+07] (Section 2.2) and reactive behavior on the Web [BBB+07] (Section 2.3) to events on the Web (Section 2.4). With the exception of the topic of events, this chapter aims at giving brief overviews. Deeper explanations can be found in the references just given.

2.1 Data on the Web

With the Web becoming a universal information system, information and data on the Web is not only read and interpreted by humans. It is also consumed by machines and programs. Machines perform task such as filtering, transforming, indexing, or deriving new information.

Documents in the Hypertext Markup Language (HTML) are structured along their intended visual presentation as a text document. Depending on the desired layout, information about flight connections inside an HTML document therefore might be structured as a table with rows and cells or simply a list. When the only processing done by machines is a visual rendering, then this structuring is fine. When machines however should perform other tasks, e.g., finding connections, then the meaning of data in the document becomes important, e.g., which data represents flight numbers, which departure airports, etc.

The need to represent information in a more structured way is addressed by the Extensible Markup Language (XML) [B+06a, B+06b] and, more recently, also by the Resource Description Framework (RDF) [MM04, KC04].

2.1.1 Extensible Markup Language (XML)

The Extensible Markup Language (XML) [B+06a, B+06b] is a so-called generic markup language (or meta markup language), which means that, unlike HTML, it has no fixed vocabulary. It just provides a syntax from which individual, more application-specific languages, e.g., a language for representing information about flight connections, are derived. These specific markup languages are obtained by restricting the vocabulary and assigning semantics. Such application-specific languages are also sometimes called XML dialects.

An XML document consists of a prolog and a root element. Additionally, processing instructions and comments may appear inside the prolog, inside the root element, or after the root element. All application-relevant data is contained in the root element. The underlying data model of XML, formally specified in the XML Information Set [CT04], is that of an ordered tree.

Each element has a name (also called tag-name) and optionally a number of attributes with assigned values. Elements are delimited by so-called tags. For an element with name `element` and attributes `attr1` and `attr2` with respective values `value1` and `value2`, the start tag has the form

```
<element attr1="value1" attr2="value2">
```

and the end tag the form

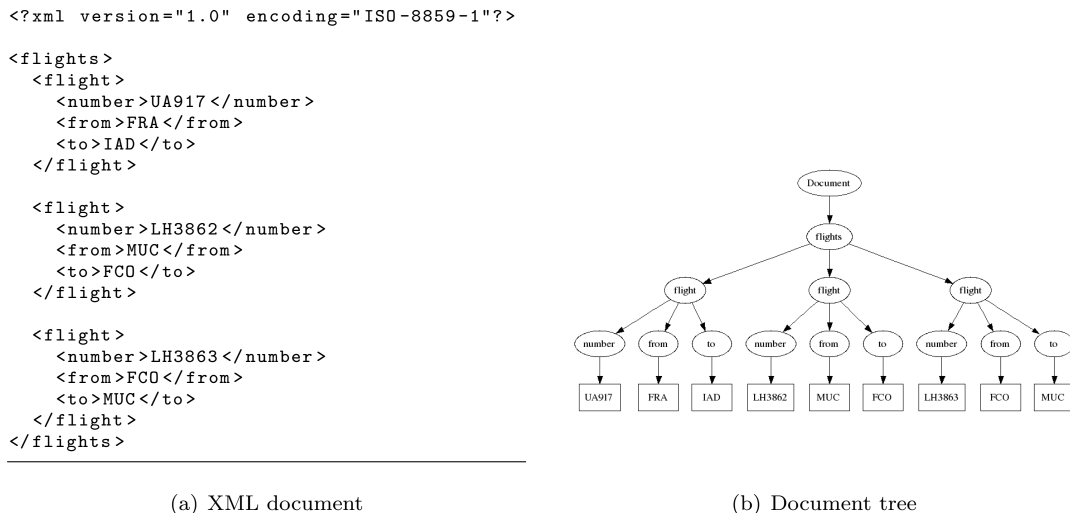
```
</element>
```

Elements can contain other elements as well as text, which we also call its children. This nesting of elements gives rise to a tree structure. The order of the children is relevant. Note that every element, except the root element, must be contained in another element (called its parent). Figure 2.1(a) depicts an example XML document containing flight information. A corresponding visual representation of its document tree structure is shown in Figure 2.1(b). In the visual representation element nodes are ellipses while text nodes are rectangles.

XML has become a popular format for exchanging information and its use reaches far beyond the Web. Its popularity can be attributed mainly to the fact that XML copes well with so-called semi-structured data, which has an irregular (sometimes recursive) and often changing structure. In contrast, the relational data model requires a rigid structure and makes structural changes difficult. By using appropriate names for tags and attributes in XML, data in XML documents also becomes self-describing.

XML has also some limitations though:

- Dealing with graph-structured data in XML can be tedious. The use of ID and IDREF attributes in an XML document, the XML Linking Language (XLink), as well as the use of element or attributes with similar, application defined semantics (e.g., `<a href="...">` in XHTML) allow to model graph-structured information in the XML tree data model.



```
<?xml version="1.0" encoding="ISO-8859-1"?>

<flights>
  <flight>
    <number>UA917</number>
    <from>FRA</from>
    <to>IAD</to>
  </flight>

  <flight>
    <number>LH3862</number>
    <from>MUC</from>
    <to>FCO</to>
  </flight>

  <flight>
    <number>LH3863</number>
    <from>FCO</from>
    <to>MUC</to>
  </flight>
</flights>
```


(a) XML document

(b) Document tree

Figure 2.1: An XML document `flights.xml` and a visualization of its document tree.

However, these mechanisms are separate from the parent-child relation in the data model of XML, and have to be resolved “manually” (e.g., through value joins or the `id()`-function in XPath) in programs and queries.

- XML offers only limited support for identifying specific nodes (elements, attributes, etc.) within a document. Elements carrying an ID attribute are easily identified, but for other elements as well as other types of nodes this is not so easy. The so-called “node identity” (typically the memory address of the object representing a node) can be used within a single query, but not for identifying nodes over a longer period of time. In particular, node identity might be completely lost if the document is updated, even if the change is minor (e.g., adding an element). Issues related to identity are also discussed in [Fur08, Chapter 3.4].
- The self-describing nature of XML aims at making documents understandable for humans but is not well-suited for making them “understandable” for machines. When seeing tags called `surname` and `last_name`, humans will easily know that the are just different terms for the same concept. For machines this is generally not possible. XML also offers no way explicitly tell a machine that these terms are semantically equivalent and should be processed in the same manner.
- Closely related, XML is purely a format for data representation. Its data model, as well as it formalisms for typing data, do not support drawing inferences such as deriving new facts. For example, XML can model a list of students together with their test scores, but it cannot make any inferences such as deriving which students passed the test. Also, the data model of XML is not suited for modeling existentially quantified information, that is, information where an object or entity is only claimed to exist without naming the concrete object or entity. For example it is difficult to express in XML that a person has a male child without explicitly specifying the child.

The Semantic Web in an effort that seeks to overcome these difficulties of XML with the aim of making Web data “meaningful to computers” [BLHL01]. Of particular relevance is the Resource Description Framework (RDF), which will be described next.

2.1.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) [MM04, KC04] provides syntax and semantics for representing knowledge about “things,” which are called resources in the context of RDF. A

resource is anything that can be identified with a URI. This includes in particular Web documents, but also objects such as physical items for sale in a catalog that are not “actually on the Web” (i.e., cannot be retrieved by means of HTTP or another protocol). A typical application of RDF is annotating a Web page with meta data like author, keywords, publication date, or licensing terms.

Resources are described by means of statements of the form subject – predicate – object, so-called (RDF) triples. The predicate is also called a *property* of the described resource, and the object the *property value*. The subject and the predicate are given with URIs. The object can be given with a URI or be a literal, i.e., a (possibly typed) constant value given as a character string such as `"IAD"`. (Blank nodes are an exception for subjects and objects that will be discussed shortly.)

Such RDF triples form essentially an unordered, directed, labeled graph. The nodes in the graph are the subjects and objects, and each RDF triple gives rise to an edge from its subject to its object with the predicate as label. Note that literal nodes cannot have outgoing edges.

In some cases it is necessary or just convenient to make statements about resources without knowing or assigning a URI for them. For example, one might want to express that there is a flight from Frankfurt to Munich without knowing a URI for that flight. To this end, RDF offers blank nodes. Blank nodes can be used instead of URIs for subjects and objects to express that there is a resource with the specified properties without actually identifying that resource.

RDF is augmented by the RDF Vocabulary Description Language RDF-Schema [BG04]. RDF Schema provides a type system for defining application specific vocabularies. By typing RDF, it both constrains RDF graphs (e.g., by expressing that “flies from” is a property of resources of the type “flight” that has a value a resource of the type “airport”) and allows simple inferences (e.g., if resource “John” is of type “employee” and “employee” is a subclass of “person” then “John” is also of type “person”). Semantics of RDF, as well as of RDF with RDF Schema, are specified formally as a model theory [Hay04].

Regarding syntax, an RDF graph can be written, stored, and exchanged in one of a multitude of so-called serialization formats. The official format standardized be the W3C is the RDF/XML Syntax [Bec04]; however there are many alternative formats. An overview over the different serialization formats can be found in [Bol05].

To summarize, RDF goes beyond XML by having directed graphs as data model, providing the ability to represent existentially quantified information through blank nodes, allowing simple inferences through RDF Schema, and providing clear semantics in the form of a model theory.

2.2 Reasoning on the Web

An important part of processing data in information systems is querying and reasoning. Typical task that involve querying and reasoning with Web data such as XML or RDF documents are:

- Selecting only relevant portions of a large volume of data: e.g., in a document about flights, select only those flights going from Munich to Rome.
- Restructuring data into new formats: e.g., transform a document about flights like the one from Figure 2.1(a) into an XHTML document that is suitable for rendering in a Web browser.
- Combining information: e.g., for each flight in a document about flight connections, find appropriate connecting hotel shuttles from a document about ground transportation options.
- Derive new information: e.g., compute multi-stop flight connections from a document containing only direct flight-connections or classify meals as vegan, vegetarian, or non-vegetarian according to their ingredients.

We now look at prevalent query languages for Web data and at approaches for more advanced reasoning and knowledge representation on the Web. Note that there is no sharp boundary between querying and reasoning.

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output
    method="xml"
    encoding="iso-8859-1"
    doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
  />

  <xsl:template match="/">
    <html>
      <head>
        <title>Flight Connections</title>
      </head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="flights">
    <table>
      <tr>
        <th>Flight Number</th>
        <th>From</th>
        <th>To</th> </tr>
      <xsl:apply-templates/>
    </table>
  </xsl:template>

  <xsl:template match="flight">
      <tr>
        <td> <xsl:value-of select="number" /> </td>
        <td> <xsl:value-of select="from"   /> </td>
        <td> <xsl:value-of select="to"     /> </td>
      </tr>
  </xsl:template>
</xsl:stylesheet>
```

Figure 2.2: An XSLT program producing XHTML output.

2.2.1 Standard Web Query Languages

The World Wide Web Consortium (W3C) defines two standard languages querying (and transforming) XML, XSL Transformations (XSLT)[1] [Cla99, Kay07] and XQuery [B$^+$07b]. Both XSLT and XQuery make use of XPath [CD99, B$^+$07a], a language for navigating the tree structure of XML documents. In its abbreviated syntax, XPath expressions are reminiscent of the way directory paths in file systems are written.

XSLT XSLT is the simpler of the two languages. As the term "stylesheet" in the name of XSLT implies, XSLT is aimed at transformation tasks that address changing the style, i.e., the way information is presented, of an XML document. Thus, XSLT is primarily intended for transformations of documents that involve simple restructuring such as renaming of elements, filtering, or moving subtrees up or down. A typical application of XSLT is transforming an XML document that uses an application-specific vocabulary (e.g., the flight connection document of Figure 2.1(a)) into an XHTML document suitable for rendering in a browser. In its output, XSLT is not restricted to XML, but can also output "old," non-XML-conforming HTML or plain text.

An XSLT program consists of templates, which specify patterns for elements in the input they match as well as output they produce. Processing of an XML document starts at the root and chooses a template with a pattern that matches the root. The output that this template produces can then cause recursive matching of templates. Typically the recursion is on the children of the

[1]XSL stand for Extensible Stylesheet Language and is more precisely a family of languages comprising XSLT, the XML Path Language (XPath), and XSL Formatting Objects (XSL-FO).

current node (however XSLT allows to specify deviations from this, where nodes for recursive matching are explicitly selected or templates explicitly called). This processing model accounts well for transformations that rename elements, insert elements between a node and children (i.e., "move a subtree down"), remove elements between a node and some descendants (i.e., "move a subtree up"), or filter certain parts of the input out.

The expressivity of XSLT 1.0 beyond these transformations is rather limited, mainly because it does not allow recursion processing where templates process the output of other templates (note that this is very different from the structural recursion over the input tree!). In XSLT 2.0 this limitation has been lifted however by adding so-called functions.

An example of an XSLT program for transforming a document like the one of Figure 2.1(a) into an XHTML document is shown in Figure 2.2.

XQuery XQuery is designed for performing more general queries and at extracting information from large volumes of XML data. XQuery is considered more expressive than XSLT and also more feature-rich (and thus considered harder to learn). Whereas XSLT is targeted at transforming a single XML document, XQuery is particularly suited for tasks requiring the extraction and combination of data (also from several sources) such as joining data, grouping and aggregating data, or reordering data.

An XQuery program essentially consists of so-called FLWOR (pronounced "flower," note however that "W" comes before "O") statements. The `for`-clause selects and iterates over nodes from the input documents. Note that a `for`-clause can cause multiple iterations, or in other words, produce a cross-product. A `where`-clause restricts the results produced by `for`-clause. A typical restriction would be requiring some equality between nodes from different sources, which essentially turns a cross-product into an equi-join. The iteration order can be changed using an `order by`-clause. Results (for each iteration) are specified in the `return`-clause.

Note that FLWOR statements are similar to the `SELECT` – `FROM` – `WHERE` statements found in SQL: the `for`-clauses (together with `let`-clauses) correspond loosely to the `FROM` part, the `where`-clauses to the `WHERE` part, the `return`-clauses to the `SELECT`-part.

XQuery has many more language features beyond the basic FLWOR statements, including the ability to define functions (with unrestricted recursion).

An example XQuery program is shown in Figure 2.3(b). From a document with flight connections (Figure 2.1(a)) and a document with information about airport shuttles (Figure 2.3(a)) it produces as output a list of "travel options" that is a list of hotels that can be reached by means of a flight and a connecting shuttle. The list is sorted by the name of the departing airport and includes travel instructions about the flight and shuttle that has to be taken. The output for the sample inputs of Figure 2.1(a) and Figure 2.3(a) is shown in Figure 2.3(c).

SPARQL For querying RDF data, the W3C has recently standardized a query language called SPARQL [PS08]. A SPARQL query either yields a set of variable bindings or constructs a new RDF graph. The former is convenient when results of a query are to be used inside some application software, the latter when an RDF graph should be transformed. Variable bindings that are either directly returned as a result or used for constructing a result RDF graph are obtained by matching so-called graph-patterns against the input RDF graphs.

Sharing many of its keywords with SQL, a typical SPARQL query has the form `SELECT` – `FROM` – `WHERE` or `CONSTRUCT` – `FROM` – `WHERE`. The `FROM` clause list the URIs of the RDF graphs that are to be accessed for this query. The `WHERE` clause specifies a graph pattern in the form of a set of triple patterns. Like an RDF triple, a triple pattern has the form subject – predicate – object; unlike an RDF triple, it can contain free variables, however. The same variable can occur in several triple patterns giving rise to the graph pattern. The graph pattern instructs the query evaluation to look for variable bindings so that the graph obtained from replacing the variables in the pattern with their bindings can be found as a subgraph in the input. The `SELECT` clause specifies to return (some of) the variable bindings as result, while the `CONSTRUCT` clause construct a new RDF graph using the variable bindings.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<shuttles>
  <shuttle>
    <name>Lufthansa Airport Bus</name>
    <airport>MUC</airport>
    <hotel>Bahnhofshotel</hotel>
  </shuttle>

  <shuttle>
    <name>Super Shuttle</name>
    <airport>IAD</airport>
    <hotel>Embassy Inn</hotel>
  </shuttle>

  <shuttle>
    <name>AAA Airport Express</name>
    <airport>BOS</airport>
    <hotel>The Tipton Hotel</hotel>
  </shuttle>
</shuttles>
```

(a) XML document `shuttles.xml`

```
<travel-options>
{
  for $flight in doc("flights.xml")/flights/flight ,
      $shuttle in doc("shuttles.xml")/shuttles/shuttle
  where $flight/to = $shuttle/airport
  order by $flight/from
  return
    <option>
      <departure-airport> {$flight/from/text()}
</departure-airport>
      <instructions>Take flight {$flight/number/text()},
                    then {$shuttle/name/text()}.  </instructions>
      <arrival-hotel>     {$shuttle/hotel/text()} </arrival-hotel>
    </option>
}
</travel-options>
```

(b) XQuery Program

```
<?xml version="1.0" encoding="UTF-8"?>
<travel-options>
   <option>
      <departure-airport>FCO</departure-airport>
      <instructions>Take flight LH3863,
                    then Lufthansa Airport Bus.  </instructions>
      <arrival-hotel>Bahnhofshotel</arrival-hotel>
   </option>
   <option>
      <departure-airport>FRA</departure-airport>
      <instructions>Take flight UA917,
                    then Super Shuttle.  </instructions>
      <arrival-hotel>Embassy Inn</arrival-hotel>
   </option>
</travel-options>
```

(c) Result

Figure 2.3: Combining information from a documents with flight connections and from a document with airport shuttles using XQuery

Beyond the basic graph matching, SPARQL accommodates for alternatives in patterns (i.e., disjunctions, keyword `UNION`), optional parts of patterns (keyword `OPTIONAL`), and filtering of results (keyword `FILTER`). By combining optional patterns and filtering (and using the `bound` property), negation and universal quantification can be expressed in SPARQL. However, other advanced queries (e.g., patterns involving search for graph nodes at arbitrary depths) are generally not a strong point of SPARQL and go beyond its expressive capabilites. Further, SPARQL does not allow any recursive inference in queries.

A criticism that is common to all three languages discussed here is that they are not versatile in the data format they accept [BFB+05]. XSLT and XQuery are suited only for querying XML data, while SPARQL is only suited for querying RDF data. However there are many potential use cases for queries on the Web that involve interwoven access to both XML and RDF. For example, a query might make use of domain-specific knowledge represented in RDF and RDFS to retrieve data from Web pages written in XML which are again annotated with RDF meta data. A Web query language that seeks to remedy this and offer versatility is Xcerpt, which we will be discussed in Chapter 4.

2.2.2 Reasoning with Deductive Rules

Query languages excel at efficiently extracting, restructuring, and combining existing data. They are not well suited for knowledge representation and reasoning tasks such as drawing inferences. However, for making machines derive new information and "understand" Web data, such inferences are needed.

Deductive rules (sometimes also called derivation rules) are a form of reasoning that has received much attention in Web and Semantic Web research. They provide a high-level knowledge representation mechanism that is both expressive and natural for human knowledge engineers.

While there are many variations of deductive rule languages, not just in syntax but especially in their semantics, the general idea is always to specify knowledge in terms of rules. A rule mirrors an if-then-sentence and states that a certain conclusion (then part) can be drawn if in the condition (if part) is met. Conditions and conclusions are usually written as logical formulas. The conclusion of a rule is also called its head and the condition its body, and rules are commonly written with the head first: *head* ← *body*. Due to the close relationship of logical formulas and queries, the body of a rule can also be understood as a query. We call these rules deductive, because they only deduce conclusions but do not modify existing data or cause any other side-effects. They are in contrast to rewriting rules, which specify transformations of terms or other data, reactive rules, which specify to perform actions, and other forms of rules.

Since they derive from First Order Logic (FOL), most current deductive rule formalisms represent facts using predicates (or relations) and terms. They therefore essentially use the relational data model with the important extension that many rule languages allow the use of function symbols in terms (whereas databases usually only allow constants). The fact that there is a flight with number "UA917" from "FRA" to "IAD" would, for example, be represented as a `flight`("UA917", "FRA", "IAD"). Rules allow to represent knowledge that derives new facts from existing facts. For example, we might know that every airline has a ticket office in every airport where one of its fight starts. In the rule language datalog (see, e.g., [AHV95]), this knowledge could be formalized as:

$$\texttt{has_ticket_office}(A, S) \quad \leftarrow \quad \texttt{flight}(N, S, L),\ \texttt{operated_by}(N, A)$$

In this rule N, S, L, and A are free variables. The comma (",") in the rule body is read as a conjunction. Importantly, most deductive rule languages allow for recursive inferences. Transitive closures such as "if there is a connection from X to Y and from Y to Z, then there is a connection from X to Z" are a typical application of such recursive inferences. The following rules realize such knowledge using our flight example:

$$\begin{array}{lcl} \texttt{connection(X,Y)} & \leftarrow & \texttt{flight(N,X,Y)} \\ \texttt{connection(X,Z)} & \leftarrow & \texttt{connection(X,Y),\ connection(Y,Z)} \end{array}$$

Because of the natural correspondence of deductive rules with English language "if-then" sentences, they are often considered to be very intuitive for humans. Further there are approaches to rule languages that do not represent rules as logic formulas but in a limited, controlled English which can be automatically parsed and translated to a representation like logic that is suitable for processing in machines.

A particular trait of rules is their compositionality. For example, the above program only finds connections using flights as a means of transportation (first rule in the program). By adding rules similar to the first rule, the program can be extended easily to include busses or trains.

As mentioned earlier, most current rule languages use the relational model for representing facts. This makes it necessary to convert Web data from its native XML, RDF, or other format, before reasoning with it. Performing such a conversion in a natural way can be difficult, though. Therefore, some rule languages aim at supporting data models that are better suited for Web data. An example is Xcerpt, which works on rooted directed graphs.

Work towards standards for Web rule languages is underway with the Rule Interchange Format (RIF) [RIF, BKPP07] at the W3C. RIF primarily uses a relational data model [BK08a, BK08b], but also strives for compatibility with RDF and the Web Ontology Language (OWL) [dB08].

2.2.3 Knowledge Representation with Ontologies

Deductive rules are not the only form of reasoning on the Web. Another form of representing knowledge and reasoning with it are ontologies. The W3C standard for representing ontologies is the Web Ontology Language (OWL) [SWM04]. It actually consists of three sublanguages of increasing expressivity: OWL Lite, OWL DL, and OWL Full. OWL is based on RDF and RDF Schema and in terms of the inferences that can be drawn increases significantly the expressiveness of RDF Schema.

OWL, like other ontology languages, is based on description logics. A description logic can be understood as a fragment of First Order Logic (FOL), usually restricted in such a way that typical reasoning tasks are, in contrast to full FOL, still decidable.[2] Ontologies model domain knowledge by specifying axioms about concepts and their relationships. (Relationships are also sometimes called roles.) Concepts correspond to unary, relationships to binary relations of FOL. Axioms correspond to FOL formulas, but are usually written in a tailored syntax that leaves variables implicit. OWL and (most) description logics do not support relations that are ternary or of higher arity, which can be a severe limitation on the knowledge that can be expressed in OWL. Also, axioms in most description can only express a tree of relationships between concepts not an arbitrary graph of relationships. (The formal reason for this is that axioms correspond to formulas in the so-called two-variable fragment of FOL, that is, to formulas that use only two variables.)

As an example consider an ontology that models knowledge about food served on flights. There are concepts such as `FLIGHT`, `MEAL`, `INGREDIENT`, and `MEAT`, and relationships such as `serves` and `contains`. Axioms can define new concepts, for example `VEGETARIAN_MEAL` could be defined as a `MEAL` that `contains` no `MEAT`. Axioms can also specify type relationships such as that `MEAT` is a type of `INGREDIENT` or more general conditions such as that every `FLIGHT` must serve at least one `VEGETARIAN_MEAL`. In typical description logic syntax these axioms would be written as

$$\begin{aligned}&\texttt{VEGETARIAN_MEAL} = \texttt{MEAL} \sqcap \forall\texttt{contains}.\neg\texttt{MEAT}\\&\texttt{MEAT} \sqsubseteq \texttt{INGREDIENT}\\&\texttt{FLIGHT} \sqsubseteq \exists\texttt{serves}.\texttt{VEGETARIAN_MEAL}\end{aligned}$$

These axioms correspond to the following formulas written in traditional FOL syntax:

$$\begin{aligned}&\forall x(\texttt{VEGETARIAN_MEAL}(x) \iff (\texttt{MEAL}(x) \land \forall y(\texttt{contains}(x,y) \Rightarrow \neg\texttt{MEAT}(y))))\\&\forall x(\texttt{MEAT}(x) \Rightarrow \texttt{INGREDIENT}(x))\\&\forall x(\texttt{FLIGHT}(x) \Rightarrow \exists y(\texttt{serves}(x,y) \land \texttt{VEGETARIAN_MEAL}(y)))\end{aligned}$$

[2]The letters "DL" in OWL DL reflect its correspondence to description logics. OWL DL and OWL Lite are decidable whereas OWL Full is not.

While ontologies might be considered more popular than rules on the Web, in part because there already is a finished W3C standard with OWL for representing ontologies, they are less important in this thesis. Ontologies are well-targeted at reasoning tasks such as subsumption (checking whether a concept A is a subset of concept B) or satisfiability (can a given set of axioms be satisfied, i.e., does it have a model). They are however not as well-suited for querying data, in particular because they are restricted to unary and binary relations and can only specify tree-shaped combinations of relationships between concepts.[3] Deductive rules, because of their close relationship with querying, are therefore considered more interesting for querying events in this thesis.

2.3 Reactivity on the Web

Queries, rules, and ontologies give a fairly passive impression of the Web: upon request certain information can be obtained by accessing data sources on the Web. However, many data sources on the Web evolve in the sense that they change over time and operations that change data are an essential part of any information system. Even a simple address book application will require at least operations to add and remove entries. An application for electronic shopping will require more and more complicated interconnected operations to realize work-flows such as order processing, shipping, and payment collection. Dealing with data sources that change over time and realizing common operations of information systems entails a need for reactivity, the ability to detect events and respond to them automatically in a timely manner.

Reactive behavior on the Web can be and has been for many years implemented using general purpose programming languages. However, higher-level reactive languages have been and are developed that aim at abstracting away network communication and system issues, at easing the specification of complex updates of Web resources (in particular XML, RDF, or OWL documents), and at being convenient for specifying complex flows of actions and reactions. In particular, languages based on reactive rules have received attention in Web-focused research communities lately [REW, P$^+$06, PKB$^+$07, BBB$^+$07].

Reactive rules are statements that formalize executable knowledge by specifying actions to be undertaken in response to certain events or situations. They are suited for both adding active behavior to content (e.g., changing an address book entry in reaction to an event signifying user input) and for formalizing content guiding active behavior of other entities (e.g., formalizing the actions of humans and systems involved in an order processing work-flow). Reactive rules revolve around events and actions and thus serve a different and complementary purpose from traditional knowledge representation using ontologies and deductive rules, which focus on drawing conclusions (implicit facts, classification in terminologies, etc.) from data.

The two most common forms of reactive rules are Event-Condition-Action (ECA) rules and production rules (also called Condition-Action rules), originating from Active Databases research [WC96, Pat98] and Artificial Intelligence research [FM77, For81, For82], respectively. ECA rules have the structure "`ON` *event (specification)* `IF` *condition* `DO` *action*" and specify to execute the action whenever an event matching event specification (which could be a complex event query) happens and the condition, which usually is a query (e.g., to a local or remote Web resource), holds. Production rules are similar but make no explicit reference to an event. They have the structure "`WHEN` *condition* `THEN` *action*" and specify to execute the action whenever the condition becomes true. In contrast to ECA rules, the condition can typically only refer to local data which is accessible in the so-called working memory. Note the seemingly minor, but semantically important, difference between the condition "being true" in ECA rules and "becoming true" in production rules.

ECA rules are often argued to be more convenient in distributed Web-based applications than production rules, since they make an explicit specification of events, which allows also for message-based communication, and have no need for a local working memory [BE06b]. Explicit specification

[3]The difficulties in ontologies and querying are also reflected in research on combining rules and ontologies, a topic that has received much attention recently (see, e.g., [Ros06] for an introductory overview).

"Event: Anything that happens, or is contemplated as happening."
"Event (also event object, event message, event tuple): An object that represents, encodes or records an event, generally for the purpose of computer processing." *Event Processing Glossary* [LS08]

"Any happening of interest that can be observed from within a computer is considered an *event*."
"A *notification* is a datum that reifies an event, i.e., it contains data describing the event." [MFP06]

"An event is a significant state change in the state of the universe. A significant state change is one for which an optimal response by the system is to take an action. An insignificant state change is one for which the system need take no action. An action may be registering information about the event in the enterprise's memory. Insignificant state changes are not registered in memory; they are never 'remembered.' " [CCC07]

"According to the dictionary, an event is a thing that happens, especially when it has some relevance. Relevance can be measured by whether some sort of action has to be taken as a result of the event. Consequently, an event can be seen as a specific situation in which one or more reactions may be necessary." [Pat98, Chapter 1]

"Messages (or 'events') flow across networks between enterprises and organizations." [Luc02]

Figure 2.4: Different definitions of "event" in literature related to event processing

of an event is also often more natural in event-driven applications. This is particularly true in applications involving CEP such as the ones discussed in Chapter 1. An extensive survey of both ECA and production rule languages is given in [BBB+07].

A concrete example of an ECA rule language for the Web will be introduced in Chapter 4 with the language XChange. Production rules will recur in Chapter 3, when discussing how they can be used to implement complex event queries. (Note that this connection of production rules and CEP, where the rules are used to *implement* complex event queries, is fundamentally different from the connection between ECA rules and CEP, where a (complex) event query (sub)language is *used* for the in the event specification).

2.4 Events on the Web

Reactivity and CEP entail the ability to detect simple (or atomic) events. We now discuss issues related to simple events. While we focus on events "on the Web," this should be understood in the broader sense including any kind of information system that is built upon Web standards whether it be publicly accessible or not. In particular it thus includes modern enterprise information systems employing Web services, but also for example Web-based Supervisory Control and Data Acquisition (SCADA) systems (see Chapter 1.1).

2.4.1 What is an Event?

A curiosity of event processing and related fields is that there can much discussion around the question of what is and what is not an event. There is no singular, commonly agreed definition as the selection of definitions from the literature in Figure 2.4 shows. However, the first two definitions clearly indicate a necessity for the representation of an event in a computer as a message, object, function call, or other entity with associated data. Since CEP deals only with these representations, much of the discussion revolving around the term "event" is rather philosophical in its context.

Throughout this thesis, we adopt this way of looking at things and equate (simple) events with their representation.

However, beyond the scope of CEP and this thesis, what is and what is not an event can be of practical relevance. The design of an event-based system includes decisions of which events (in the sense of something that happens) are given a representation that is then exposed to other components. Essentially the types of events a component exposes become part of its external interface. In this context, events are subjected to typical issues and discussions of interface design such as responsibility (e.g., should the database or the application software signal that an address has been changed) or granularity (e.g., one single event for an address change or one event for each line in an address).

Additionally, there might be an issue how to detect an event that happens in a given application when this application does not provide explicit support for the event, i.e., it does not explicitly signal this event to the outside world, e.g., by sending a message. As an example consider a Web site on which the event that its content changes should be detected. Most Web sites are simply changed without providing a mechanism that signals this change to potentially interested parties.[4] An option for an interested party in this case might be to periodically retrieve the Web site and compare it with a previously stored version of the Web site to detect any changes. Issues relate to detecting such events are beyond the scope of this thesis; however we will discuss how this issue might related to CEP and complex event queries in Section 19 on research perspectives.

2.4.2 Data Formats and Models for Events

Since event-based systems and in particular CEP applications deal with representations of events, their design is influenced by the way events are represented. It is common to represent events and their associated data using existing data formats and models, e.g., as XML documents, Java objects, or relational tuples. Such a representation of events as ordinary data is particularly necessary when events must be communicated in a distributed system.

When events are transmitted over the Web as messages, then these messages are typically XML documents. Note however, that the representation used for transmitting events is not necessarily the same as the representation used for processing it by sender or receiver of the event. For example, an event might be created as a relational tuple at the sender, this tuple then be serialized into an XML message for transmission, and the XML message be de-serialized into a Java object by the receiver. Also, an event might have multiple receivers and these might use different representations. For example an order processing application might convert incoming XML messages about order events into objects for processing them, while a monitoring application receiving the same events might process the XML messages directly.

The following data models and data formats are commonly used for representing events:

- **Typed events without data**: an event is represented as a single symbol that signifies its type but has no further data associated with it. Examples would be "$AlarmFromSensorA$" and "$AlarmFromSensorB$" to signal alarms from sensors A and B respectively. This representation is fairly limited and works well only for very simple applications.

- **Typed events with attributes**: an event has a type and provides data in the form of named attribute-value pairs. An example would by "$Alarm\{sensor = ``A'', temperature = 42\}$" to signal that a sensor with the name "A'' raises an alarm with a measured temperature of 42. Note that there still remains an open question regarding the data model of the values; these could be just atomic values (integers, strings, etc.) —in which case it is similar to the relational model— or also list- or set-based values. Further there might be a requirement that all events of the same type provide the same attribute or not.

- **Relational tuples**: an event is represented with a symbol denoting the type of the event and values for attributes. All events of the same type must have the same attributes. These

[4] Note however, that there are some Web sites that will allow users to register so that they will receive an e-mail informing them about changes.

attributes and their respective types are typically specified in advance in a schema. Further all attribute values are usually atomic. Attributes might be identified by their position or by name. An example would be $Alarm("A", 42)$ to represent the event from earlier and using positions to identify attributes.

- **Objects**: an event is represented as an object of some object-oriented programming language (e.g., Java, C++). To write such objects in a human-readable form, the notation of typed events with attributes is commonly used. However using objects gives rise to aspects usually not covered by the previous models. Object-oriented systems support type inheritance and thus types are not mutually exclusive; an object of type A might be also of type B, e.g., if class A inherits from class B. Attribute values in objects might be references to other objects, including other event objects. When references are used, this means that the event data is not contained just in the event object but also other external objects, which might be modified over time. Also references raise a number of issues in distributed systems when an object refers to a remote object. Finally, objects can have methods associated with it to query or modify the state of the object.

- **XML messages**: an event is represented as an XML message. An example might be

```
<?xml version="1.0"?>
<alarm>
  <sensor>A</sensor>
  <temperature unit="Celsius">42</temperature>
</alarm>
```

  to represent the sensor event from earlier. The data model of XML can be seen as a labeled, ordered tree. In contrast to the previous representations of events, XML provides not only a data model but also and primarily a serialization format. This is particularly relevant for transmitting events in a heterogeneous, distributed system such as the Web.

Because XML is the primary format for exchanging information on the Web, it can be expected to become also the primary format for representing events on the Web. In particularly, other data formats are often serialized into XML; in the case of objects, there is often significant tool support for doing serialization and de-serialization without much manual programming work.

There are a number of XML-based envelope formats relevant for transmitting events. They are called envelope formats because they leave application-specific content open and focus on issues that are not application-specific such as routing of the message from the sender to the receiver(s) or metadata associated with an event.

The most popular envelope format it SOAP [G^+03]. A soap message consists of a header and a body. The header contains information related to issues such as routing and relaying of messages as well as message exchange patterns. The body contains application-specific XML content. SOAP primarily focuses on messages, not events and as such does not address meta-data that is typically associated with events such as occurrence time.

The Common Base Event (CBE) format [IBM04] is an XML format tailored for events. It focuses on providing fields for meta-data typically associated with an event such as its origin, its occurrence time, or its severity.

There are also some more application specific XML message formats. For exchanging messages in Supervisory Control and Data Acquisition (SCADA) Systems, the Facility Control Markup Language (FCML) [BLO^+08] has been developed. Note that currently FCML is used in a context where a system actively requests data from sensor rather than where sensors proactively send data to the system.

So far, there has not been much work on using RDF to encode events. It is of course possible to communicate RDF information in XML messages using its XML serialization. However, processing the XML serialization of RDF rather than RDF in its native data model is very inconvenient, so that RDF should be considered as data model in its own right not part of XML. With more

and more information on the Web being represented as RDF, we can expect RDF data in events to become relevant in the near future. Particularly, many scientific data sets (e.g., biological databases) use RDF, and a change notification service might generate events that contain the changed data as RDF.

2.4.3 Communication of Events

As with all XML messages on the Web, events are primarily communicated between different Web nodes using HTTP. A message can be sent to a Web node identified by its URI using either the GET or the POST method of HTTP, but POST is preferable. When GET is used, the XML message must be encoded as part of the URI and is then often subject to length restrictions. Further, GET requests should be idempotent [F$^+$99], i.e., when the same GET request is repeated it should not have any further side-effects, which cannot always be assumed for the reception of an event.

It is also conceivable to use other Internet protocols for communicating events. For example, SOAP messages might also be transported using e-mail protocols, in particular the Simple Mail Transfer Protocol (SMTP) [MKW$^+$02]. This is particularly interesting in cases where the recipient of an event might not be connected to the Web all the time. E-mail infrastructure will then provide for storing the message until the recipient comes online and will retrieve e-mails.

Communication on the Web is, as in general for distributed systems, not reliable. When sending a message, the sender cannot be sure if the recipient has received the message or not. Only by having the original recipient sending an acknowledgment message back, the original sender can be assured. However, in this case, the original recipient (and sender of the acknowledgment) can now not be sure that the original sender (and recipient of the acknowledgment) has received the acknowledgment. This unreliability causes problems when distributed nodes must agree on a single common state, especially with distributed transactions. There are a number of solutions to these issues such as a two-phase commit protocol (see, e.g., [CDK01]). While relevant in the overall context of an event-driven application, these issues are not directly to CEP and will thus not be detailed further.

2.4.4 Timing and Ordering of Events

Timing and order of events can be relevant for the processing of events and are of special interest in CEP. However, timing and order are difficult issues in distributed systems. Each node (computer, device, etc.) in a distributed system has its own local clock and the clocks of different nodes are not and cannot be perfectly synchronized [CDK01]. The time it takes to transmit a message varies depending on the sender and receiver, the route taken between them (if there are several), network traffic, and other factors. These issues are somewhat related, because the precision that can be reached when synchronizing clocks through message exchange is limited by the transmission times of messages.

There are three main issues related to timing and ordering of events in distributed systems such as the Web: assigning time stamps to events, logical order of events, and arrival order of events. A deeper introduction into timing and ordering issues in distributed systems can be found, e.g., in [CDK01].

Time stamping In event processing applications it is common to have temporal constraints on events such as "event A must happen before February 20, 2009" or "events A and B must happen within 5 minutes of each other." This requires that events are assigned a time stamp signifying their occurrence time, typically at the event source. In a distributed system, however, each nodes has its own local clock and the different clocks are not perfectly synchronized. When events from different sources are processed, then the time stamps of events are given according to different clocks and thus comparisons and computations subject to imprecision.

When a synchronization protocol such as the Network Time Protocol (NTP) is employed in a distributed system, the drift between different clocks will typically be in the order of tens of

milliseconds [CDK01]. In many real-world applications, this imprecision can simply be tolerated. For example, when there is a constraint like "an order must be shipped within 24 hours," then it can be expected that the processes generating these events will aim at shipping most orders within a significantly lower time and not reach 24 hours. For those few events that come very close to the 24 hours, the decision whether they are within or outside might be somewhat arbitrary.

Such arbitrary decisions relating to unsynchronized clocks are a fact of everyday life, and clocks (watches, wall clocks, etc.) used by humans are often subject to much greater imprecision than clocks in computer systems. That such imprecision is often tolerated is also evidenced by the fact that temporal constraints are often specified using time units larger than second: these time units do not have a fixed length and their interpretations might vary. A minute might last 60 or 61 seconds, the latter being the rare case when it contains a leap second. A distance of one month between two events might be interpreted as the second event happening on the same day in the next month, e.g., if the first event happens on February 20, then the second must happen on March 20. (This also has a potential problem if this day does not exist in the next month, e.g., there is no February 30.) A distance of one month might also be interpreted as being 30 days or some other length.

On the Web there is an additional problem that clocks might not be synchronized to a known precision and, in particular, an event source might maliciously assign a time stamp that is earlier or later than the actual time. To give an example, the date and time assigned to an e-mail are given according to the sender's clock which might diverge significantly from the actual time either accidentally or deliberately. For Web applications that are sensitive to time (e.g., online auctions) and that receive data from untrusted sources (e.g., bidders in an auction), it is therefore common to time stamp event (e.g., bids) only at the event receiver (e.g., the auctioneer). Note that this is also an issue common in everyday life: for messages that are associated with a certain deadline such as job applications or entries into a lottery, the time assigned to the message is usually given by the recipient (typically date of the reception of the letter) or a trusted third party (typically the postal date stamp).

Logical order It is also common in event processing that the order of events is relevant, e.g., that "event A happens before B." While the imprecision of time stamps can usually be tolerated when measuring distances between events, it makes time stamps in some cases ill-suited for ordering events. Events can happen in very rapid succession with the distance between two event being smaller than the imprecision associated with clocks in distributed systems. Accordingly when events from different sources are ordered according to their time stamps, this order might not mirror the logical order in which the events happened.

The logical order of events in a distributed system is determined by potential causal relationships between the events based on the following principles:

- If two events happen at the same node, then their logical order is determined by the local clock.
- If a message is sent between two nodes, then the event of sending the message happens before the event of receiving the message.
- The potential causal ordering is transitive, i.e., when a happens before b and b before c, then a happens also before c.

The partial order obtained from these principles is called the potential causal ordering (also causal ordering or happens-before relation). Note that in contrast to an ordering based on time stamps, this is only a partial order not a total order.

Lamport logical clocks [Lam78] are a mechanism for capturing the potential causal ordering numerically. Each node in the distributed system has a logical clock L_i that is incremented whenever a new event is generated. When two nodes i and j exchange a message, the sender i also sends its clock value L_i with the message. The receiver j resets its clock L_j to the maximum of its current value of L_j and the received value of L_i. These logical clocks guarantee that if event

a happens before event b, then also the time stamp of a (given according to the local clock of its source) is smaller than the time stamp of b. Note that the reverse does in general not hold. The time stamps given by Lamport logical clocks do not introduce a total order on events because event might have the same time stamp value. However by using an arbitrary but fixed order on the nodes, it can be turned easily into a total order.

Note that mechanisms for determining the order of events such as Lamport logical clocks require a cooperation of nodes in a distributed system. On the Web, such a cooperation and the necessary trust between the involved parties cannot be assumed in the general case and only work, e.g., in closed intranets.

Arrival order In event processing, there is a third relevant order for events, the order in which events (more precisely the notifications about them) arrive at some receiving node. This order might not correspond to the time stamp ordering or the logical order of events. The reason for this is that transmission times for messages between different nodes might vary significantly.

Rather than discussing the effect of varying transmission times in term of the milliseconds typical in computer networks, it is best to look at an example outside computer systems with longer, "exaggerated" transmission times to explain their effect. Consider a mail order company. An incoming order (event) is received at the order department. From there it is forwarded by in-house mail to the billing department and by fax to the shipping department. The shipping department ships the products almost immediately and sends a notice about the shipping (event) to the billing department. Logically, the order event happens before the shipping event, and typically this will also be true for the time stamps assigned in the order and shipping department. However, in the billing department, the arrival order is reversed: since in-house mail takes much longer than fax, the shipping notice arrives before the order notice. This reversed order can significantly affect the way events must be processed at the billing department. For example, the billing department might watch out for shipping events that are not preceded (logically) by a corresponding order event. This cannot, however, be checked immediately when the shipping event is received.

We can see from this example that the arrival order of events at a given event processor might not correspond with either the time stamp ordering or the logical ordering. However, there are typically upper bounds on the transmission times. We can use these upper bound to re-order events upon arrival. For example, we might know that in-house mail takes at most one business day. Accordingly, the corresponding order for a shipping notice received by fax can be delayed at most by one business day. To check whether a shipping was not preceded (logically) by an order, one might therefore have to wait at most one business day.

Chapter 3

State of the Art

Complex Event Processing has only recently emerged as a discipline in its own right. However, CEP has many independent roots in various fields, including (see also [Luc08]):[1]

- pattern detection in discrete event simulation, e.g., of hardware designs, control systems, and factory production lines (early 1990's [GL92, LVB+93]),
- monitoring and intrusion detection in computer networks, usually as part of network management (roughly 1990's [SR90, MSS97]),
- composite event detection in active database management systems (mainly in the early 1990's [GJS92a, GD93, CKAK94])
- message-oriented middleware for distributed systems (late 1990's [HBBM96, SSS+03]) and later on a higher abstraction level event-driven architecture (roughly since mid 2000's [Etz05]),
- temporal representation and reasoning in Artificial Intelligence (roughly since the 1980's [All83, KS86], but with much earlier roots [MH69]).

Developments in each of these areas proceeded often in parallel and with little exchange among the communities. Accordingly, there is a very diverse terminology, a wide range of approaches, and different expectations and requirements in terms of functionality of CEP. For example, the terms "complex event" and "Complex Event Processing" originate from David Luckham's work in the area of discrete event simulation. Research in the active database community, which took place mainly around the same time, used the terms "composite event" and "composite event detection" instead. In work on data streams processing finally, there are usually just tuples, streams, and (continuous) queries and no designated terminology for complex events.

This section surveys the state of the art in CEP. Since CEP is a field that is very broad and without clear-cut boundaries, this section focuses strongly on the topic of this thesis, that is, on querying complex events. It concentrates on languages and formalism for detecting complex events that are known and specified a priori. Other, less developed aspects of CEP, for example detecting unknown complex events using approaches like machine learning and data mining on event streams, are not discussed here but in Chapter 19 on future perspectives of CEP.

3.1 Querying Complex Events Unraveled

Many different languages for querying complex events have been developed in the past. To the best of our knowledge, there are no other comprehensive surveys that compare the designs of different

[1] The given time frames given in parentheses are only rough estimations. The time frames are based on representative publications that are cited here. Note that they are supposed to show when CEP-related work was done first in the field. For example, discrete event simulation is a field as old as the 1950's, but work explicitly related to CEP happened mainly in the late 1990's.

event query languages so far. Both the multitude of languages and the lack of comprehensive surveys can be attributed in part to the fact that CEP has evolved in many independent roots and is only now recognized as field in its own right.

3.1.1 Three Styles of Querying Events

To bring some order into this multitude of event query languages, we try to group languages with a similar "style" or "flavor" together. We will focus on the general style of the languages and the variations within a style, rather than discussing each language and its constructs separately. It turns out most approaches for querying events fall into one of the following three broad categories: languages based on composition operators (also called event algebras), data stream query languages, and production rules. As we will see, the first two approaches are explicitly languages dedicated for *specifying* event queries, while last one is only a clever way to use the existing technologies of production rules to *implement* event queries. The language XChange^{EQ} that is developed in this thesis introduces a fourth style where event queries are specified in a tailored event query language that is somewhat reminiscent of logical formulas. Chapter 17 will contrast this style with the others.

The three main styles for querying events are introduced in Sections 3.2, 3.3, and 3.4 and compared in Section 3.5. Finally, we discuss some hybrid approaches that attempt to combine different styles in Section 3.6.

3.1.2 Terminology and Basic Functioning of Event Queries

Before diving into matters it is worth to recall some terminology and the basic functioning of event queries, and to make it more precise. Event queries are evaluated over time in a system called the *event processor*, event query evaluation engine, or CEP engine. Its *input* are data objects called *simple events*. Recall that we equate simple events with their representation as message or other data object here.

Like for other types of data, it is common to classify event according to some *type*. The type can be given through the schema in the case of events represented as XML message, through the class in the case of events represented as objects, or as an explicit type name in the case of events represented as tuples in a relational data model (the type in the case is the name of the relation or predicate the tuple belongs to). In general, a given event can belong to several types, e.g., by means of inheritance in object oriented models or by conforming to multiple schemas in XML.

Since simple events are received over time, the input takes the form of one or more *streams*. Every simple event is associated with at least one time point or interval called its *occurrence time*. Unless otherwise specified we assume here for simplicity of presentation that events have only one occurrence time. Where issues related to timing and order of events (cf. also Chapter 2.4.4) affect languages, this will be pointed out appropriately.

The event processor has a number of running event queries that specify interest in certain patterns of events in the stream(s) of incoming events. Occasionally the event processor detects a new answer to one of its queries. These answers are also called *complex events*. Note that the term "complex event" always implies that there is a query. Like simple events, complex events are always associated with a time stamp.

The *output* of the event processor are these query answers, or complex events. The output typically takes the form of an new data object (e.g., a message to be communicated to another system). However, the term output should be understood broadly to include for example also cases where the event processor does not explicitly construct new data but directly initiates some action, e.g., an update to a database or displaying something in a graphical user interface.

In order to evaluate its queries, an event processor has to maintain some information about which events have been received so far in its input streams. Events that have been received up to a given point in time are called a *complete history*. Often, an event processor will not store a complete history of events but only a *partial history*, that is a selection of the events that have been received up to a given point in time.

3.2 Composition-Operator-Based Event Query Languages

The first group of languages that we discuss builds complex event queries from simple event queries using composition operators. Historically, these languages have their roots primarily in Active Database Systems. Some examples include: the COMPOSE language of the Ode active database [GJS92b, GJS92a, GJS93], the composite event detection language of the SAMOS active database [GD93, GD94], Snoop [CKAK94] and its successor SnoopIB [AC05, AC06], the language proposed in [Ron97], GEM [MSS97], SEL [ZS01], the language in [ME01], the language in [HV02], Amit [AE04], the language in [CL04], CEDR [BC06], the language in [BKK04], ruleCore [SB05, MS], the language in [SSSM05, SSS^{+}03], the SASE Event Language [WDR06], and the original event specification language of XChange [Eck05, BEP06a, BEP06b] (which is now being replaced by XChange$^{\text{EQ}}$).

Such languages are also often called event algebras because they can be thought of as a set (the simple events) and some operations on it (the composition operators). However the term "algebra" gives many associations that often do not apply to these event query languages like a restriction to binary operators, a richness in laws for restructuring expressions (e.g., associativity, commutativity), neutral and inverse elements, or closure (i.e., the result of an operation is a member of the base set). Also, the term "algebra" is often strongly connected with query evaluation and the way queries are expressed on the user-level need not have much to do with that. We therefore avoid the term event algebra here and talk more generally of composition-operator-based languages.

3.2.1 General Idea

To explain composition-operator-based languages it is conceptually convenient to imagine all simple events being received in a single stream. The stream contains events of different types, i.e., when events are represented as XML messages, their schemas may be different, when events are represented as objects, their classes may be different, and when events are represented as tuples or collections of attribute-value pairs, their associated types may be different.

So-called simple event queries serve the purpose of specifying a certain class or type of simple events such as "order" or "payment notification" in the stream of all incoming events. Accordingly, these simple event queries primarily address the type of an event and are also often called event types in the literature. A simple event query can be understood as taking the stream of all incoming events as input and producing as output a sub-stream containing only events of the specified class or type. Additionally, simple event queries also sometimes extract data from events (such as the account number and amount for a payment notification), typically by binding variables. It should be mentioned that in many event query languages based on composition operators, especially the early ones, data in events has been treated with some neglect. When there is no data associated with a simple event queries, we will write it as a single identifier (A, B, $order$, etc.).

Composition operators can be understood as functions whose input and output are streams of events. Typical examples of composition operators are a binary conjunction and a binary sequence. They take as input the two streams which are produced by their arguments, and produce as output a single stream of composed events. Conjunction is often written $A \wedge B$ and informally means that events of types A and B must happen (at different times), regardless of their order, to yield an answer for this operator. Sequence is often written $A; B$ and additionally gives the constraint the that A event must happen before the B event in time. There are also ternary operators, e.g., a negation to detect a sequence of A and C where not B event happens between them. Although this is often written $A; \neg B; C$, this must be understood as a ternary operator not an expression formed with two binary sequences and a unary negation.

Composition operators allows to write common queries in a very compact form. With the exception of very few languages, expressions formed with operators can be arbitrarily nested. For example $(A \wedge B); C$ would detect a conjunction $A \wedge B$ being followed by an event of type C. This nesting gives rise to more expressive queries. However, it also causes problems and misunderstandings. Some equivalences that one would except such as $(A \wedge B); C \equiv (A; C) \wedge (B; C)$

or $(A;B);C \equiv A;(B;C)$ do not hold, or worse, do or do not hold depending on varying operator semantics.

3.2.2 Sequence: One Operator, Many Semantics

The sequence operator $A;B$ seems rather intuitive, it just detects complex events consisting of an event of type A followed by an event of type B. A closer look however reveals that there are at least six different semantics for the sequence operator [ZS01]. These are obtained by varying the operator in two aspects: first in three ways of interpreting what it means that an event is "followed by" an event, second in two ways of defining occurrence times of complex events.

An event a being followed by another event b can be interpreted as:

- the occurrence time of a being temporally earlier than that of b without any further restrictions,
- the occurrence time of a being temporally earlier than that of b and there being no event between a and b that is relevant to the query (e.g., if the query is $(A;B) \wedge C$ then there must not be an event of C between a and b, but there may be events of a type D that is not A, B, or C),
- the occurrence time of a being temporally earlier than that of b and there being no event whatsoever between a and b.[2]

The latter two case can also be thought of *a immediately* following b. A related issue is also discussed in [WRGD07], where a "next" operator that identifies a *single* event that immediately follows a given event is defined and its computational complexity analyzed. (Note that the above variations can have multiple immediately following events, esp. in cases where several events happen at the same time.)

Occurrence times can either be

- time points, which implies that the occurrence time of an answer to $A;B$ is the same as the occurrence time of the B event, or
- time intervals, which implies that the occurrence time of an answer to $A;B$ is the time interval that has as start the start time of the A event and as end time the end time of the B event.

Using time points or time intervals can have far reaching consequences. Consider a query $A;(B;C)$ under the first interpretation of "followed by" (i.e., non-immediately) and the events b, a, c arriving in that order and being of types B, A, and C, respectively. Under time point semantics, they yield an answer to the query: b and c satisfy $(B;C)$ with the occurrence time (point) of c, which in turn is later than that of a. Under time interval semantics, the events do not yield an answer to the query: b and c satisfy $(B;C)$, but their occurrence time is the interval starting at b which is *not* later than a.

The semantics of operators also have consequences on equivalences between different queries. It seems intuitive that a sequence operator is associative, i.e., $A;(B;C)$ and $(A;B);C$ are equivalent. The example just used to contrast time point and time interval semantics shows however that this is not the case under time point semantics, though it is the case under time interval semantics [GA02].

Since time interval semantics seem more natural for complex events and have the intuitive equivalences, most recent composition-operator-based languages therefore adopt time interval semantics. Also there is a trend to interpret sequence as non-immediately following events, i.e., other events might occur in the meantime. In many languages, an immediate sequence can still be expressed using an explicit negation of the events that must not happen in the meantime.

[2] Reference [ZS01] omits this last case and thus arrives at only four instead of six possible semantics for the sequence operator.

The binary sequence operator is often generalized to an n-ary version $A; B; C; \ldots$. This version can be reduced a nested binary sequences $(((A; B); C); \ldots)$. Note that the parentheses must be placed in this way in case the sequence operator is not associative (see above).

3.2.3 Core Composition Operators

Composition-operator-based languages offer a considerable amount of different operators. At the core of all languages are four operators: sequence, conjunction, disjunction, and negation. The sequence operator has already been discussed in detail, and we now discuss the other three core operators.

Conjunction A conjunction $A \wedge B$ specifies that both events of both types A and B must happen. It is important to emphasize that conjunction is "temporal," that is the events may happen at different time points. Since the order of events is irrelevant, conjunction does not have several semantic variants like the sequence operator. Because conjunction is "temporal" (as opposed to "logical"), some equivalences that one might intuitively expect do not hold. For example one might expect that $(A \wedge B); C$ could be equivalently written as $(A; C) \wedge (B; C)$, which is not case. Consider events a, c_1, b, c_2 of types A, C, B, and C arriving in that order. The first expression gives on single answer using a, b, c_2. The second expression gives two answers, one $(A; C)$ is matched with a, c_1 and one where $(A; C)$ is matched with a, c_2. An n-ary version of the binary conjunction is straightforward.

Disjunction A disjunction $A \vee B$ specifies that an event of type A must happen or an event of type B. Semantics of disjunction can vary when both events happen. Consider a query $A; (B \vee C); D$ and events a, b, c, d of types A, B, C, D arriving in that order. The query could either yield one result a, b, c, d or two separate results a, b, d and a, c, d. Note that the former option might be considered inconsistent with the fact that $A; (B \vee C)$ must delivers two answers (because the query cannot know that b will be followed by c when processing b). Again, an n-ary version of the binary disjunction is straightforward.

Negation Detecting the absence of events is important in many CEP applications, in particular to detect malfunction, unavailability, or delayed processing. To this end, composition-operator-based languages support a negation operator. Since the stream of incoming events is potentially unbounded, absence of events makes sense only on a restricted finite extract, or "window," of the stream — otherwise the query would have to wait for the end of the stream, potentially forever, to be sure that the event will not come.

Typically, negation is a ternary operator with the first operand specifying the start of the window, the second the event that must be absent, and the third the end of the window. Although it is often written as $A; \neg B; C$, note that negation really is one ternary operator, not a combination of two sequence operators with a unary negation operator. In languages where (complex) events can be time intervals, such interval events also be used for specifying the window (rather than two time point events for start and end). An example are the original event composition operators of XChange [Eck05, BEP06b], where the negation operator then would be written as `not` A `during` B (B being a complex event query).

3.2.4 Advanced Composition Operators

While the four operators discussed so far, sequence, conjunction, disjunction, and negation, are at the core of virtually all languages based on composition operators, there is a multitude of further operators that have been proposed in different languages. While it can be considered a strength of the composition-operator-based approach that new operators can be added easily, this also shows a weakness: there is so far no commonly agreed upon set of composition operators that offers a sufficient expressivity for most CEP applications. We now look at a selection of further operators.

Temporal Relation When events occur over time intervals, sequence is not the only temporal order in which events can happen. The occurrence time intervals of two event can also overlap, contain each other, etc. There are 13 such relationships possible [All83]: `before`, `contains`, `overlaps`, `meets`, `starts`, `finishes`, `equals`, and the respective inverses `after`, `during`, `overlapped by`, `met by`, `started by`, `finished by`. (For `equals`, the inverse is again `equals` itself.) The language in [Ron97] proposes composition operators based on these relationships.[3] The sequence operator (under the "non-immediately following" interpretation) is the same as the `before` operator in this case.

Note that there is an important difference between a temporal relationship like `before` and the corresponding composition operator, though: as a temporal relationship it is simply a function mapping two time intervals to a truth value; as a composition operator it is a function mapping two input streams of events to an output stream of (complex) events. As will be discussed in Chapter 17 this difference is important and can lead to misinterpretations of the operators. Unlike the sequence operator, there are usually no n-ary generalizations of the binary operators.

Not all of the 13 relationships are equally important. Due to timing issues (see Section 2.4.4), two different events rarely happen (or begin or end) at exactly the same time. The operators `meets`, `starts`, `finishes`, `equals` (and their inverses), however, require some endpoints of the occurrence time intervals of events to be exactly equal. These operators can therefore be considered less useful than the other operators `before`, `contains`, `overlaps` (and their inverses).

Metric Temporal Restrictions Temporal relations are qualitative in the sense that they only affect the relative order of events. Event queries however also require quantitative or metric statements such as that two events happen within a specified length of time. On unbounded event streams, such metric constraints are important to restrict queries in such ways that their evaluation can remove every stored event after a finite, fixed amount of time.

Some languages introduce temporally restricted variations of their operators to this end [HV02, CL04]. For example $(A;B)_{2h}$ would specify that A and B happen in sequence and within 2 hours. Alternatively, some languages that are based on time interval events offer a single unary restriction operator C `within` $2h$ a that restricts the duration of a complex event [BEP06b, Eck05]. Note that through nesting of expressions, this single unary operator gives essentially the same effect as having different variants of operators. For example $(A;B)_{2h}$ can be expressed be letting $C := (A;B)$, i.e., $(A;B)$ `within` $2h$. Such a unary operator however only makes sense on time interval events.

It is worth noting that these temporal restrictions offered by composition operator-based-languages are fairly limited in their expressivity. A query specifying that A happens, then B happens within 1 hour of that A, and then C happens within 1 hour of that B, cannot in most languages based on time intervals.[4] Note that neither $((A;B)_{1h};C)_{1h}$ nor $(A;(B;C)1h)_{1h}$ solves the query correctly. Both would require C to happen within 1 hour of A (not B!). It would be possible to define an n-ary temporally restricted sequence operator along the lines of $A;_{1h} B;_{1h} ; C$ that can solve this query. We are however not aware of a language that offers such an operator.

In addition to the relative restriction that events happen within a given length of time, some event query languages also allow absolute or periodic restriction. An absolute restriction would be that event A and B happen within a fixed time interval, e.g., a single day that given by its date. A periodic restriction would be that event A and B happen in the same time interval of a given sequence of (usually non-overlapping) time intervals. An example would be that events A and B both happen on a Monday (but both events on the same Monday!). Such a periodic restriction is found, e.g., in [GD93].

Temporal Events Temporal events are events that are not represented in the stream of incoming events but are generated at times specified in the query. One can distinguish relative and

---

[3] The inverses are left out in [Ron97], since instead of A `after` B, one can simply write B `before` A. Further, `before` is called `precedes` in a deviation from the original terms of [All83] there.

[4] For languages based on time points, the same argument still can be made with a variant of the given query that specifies that C happens within 2 hours of the A instead.

absolute temporal events. Relative temporal events are defined relative to another event, e.g., an event "1 hour after A" would be relative to the event A. Relative temporal events are important especially in connection with negation. For example that a request A has been "timed-out," i.e., not received its answer B within 1 hour would be specified as a negation of B between event A and the relative temporal event $A + 1h$: $A; \neg B; A + 1h$. Note that the syntax $A + 1h$ (which mimics [GD93] and [MSS97] amongst others) is ambiguous in case an event query contains the event type A multiple times.

Absolute temporal events specify fixed time points or intervals. They can be non-periodic, i.e., a single time point or interval, or periodic, i.e., a sequence of time points or (usually non-overlapping) intervals. Absolute temporal events are fairly uninteresting for event queries, because they happen only once and accordingly they become irrelevant on unbounded event streams after finite time. Periodic events have some applicability for collecting events, e.g., collecting all stock price events on a given trading day.[5] Arguable, one can claim that absolute temporal events are not a composition operator in the narrower sense, rather a primitive event (query). However there are also cases where they are explicitly operators, e.g., with the periodic operators of Snoop [CKAK94, AC06], in which case they sometimes also realize further functionality such as collecting event in a time interval.

Counting An operator that seems simple but turns out to have hidden difficulties is counting of events. Counting of events of the same type (e.g., login attempts with incorrect passwords) and generating complex events when the number exceeds a certain threshold (e.g., three such attempts generate a notification to the administrator) is not uncommon in some CEP applications.

Such operations are supported in many languages [CKAK94, ZU96, AC06, ZS01, ME01, SB05, BC06].[6] However, there are many variations in the precise semantics of the different operators. The variations come into play mainly when looking at what happens after the first answer has been generated and when another event of the type that is counted is received (e.g., a fourth attempt to login with an incorrect password). Some operators will generate a single new answers, some several new answers (e.g., three answers, one for each combination of three login attempts that uses the fourth), some will not generate an answer at all (e.g., only when again three new attempts have been made after the first answer).

Some languages offer also operators that specify that a threshold may not be reached ("at most n events") or must exactly be reached. Like negation, these queries make only sense over a finite window on the unbounded stream of incoming events.

Still More Operators Though many operators have been discussed so far, there are still more to be found in the literature. Some operators that will not be elaborated on here include: repetition in the sense of a closure operation that collects events of the same type (e.g., $*$-operator in [GJS92a, MZ97], `collect` in [Pǎt05]), the periodic and a-periodic operators of Snoop and SnoopIB (`A`, `A*`, `P`, `P*` in [CKAK94, AC06]), selection of only certain occurrences such as only every n-th event (e.g., `every` n in [Pǎt05]), and extended versions of counting that allow events of different types (e.g., `ANY` in [CKAK94, AC06, GA02], m `of` in [Eck05, BEP06a], `atleast`, `atmost`, n`th` in [AE04]).

3.2.5 Detection Time of Complex Events

Complex event queries are associated with actions that are to be performed whenever a complex event is detected. Since actions are sensitive to timing and ordering, it is therefore important to know *when* a complex event is detected (and thus an action executed) in order to understand the behavior of an overall system.

[5] Note however that data stream systems (Section 3.3 of this chapter are often better suited for such queries involving collecting events in order to perform aggregations (average, minimum, maximum).

[6] Counting operators are fairly similar to operators selecting only every n-th occurrence of an event type as they can be found, e.g., in [GJS92b, GD94, HV02]

The usual way, which works well for the core operators, is to detect a complex event when the last contributing event happens, i.e., when the last event required to produce a new answer to the complex event query happens. This last event is sometimes also called the terminator. (In analogy, the first event is sometimes called the initator.) However, for some queries using advanced operators, it is not alway clear what the last event actually is.

Especially when visualizing the time intervals, it might seem that a complex event answering the query A `overlaps` B occurs over the time interval which is the intersection of the intervals for A and B. Accordingly, one would expect the detection at the end of the time interval for A. This is, in fact, the way the `overlaps` operator had been originally defined in [Ron97]. As is pointed out in [GA04], however, the detection of the complex event cannot happen until end of the time interval for B. During the complex event detection we cannot know that B actually happens (and thus the complex event happens) until the detection of B is completed. For example in the case that $B = C; D$ we have to wait if a D event ever arrives. A similar argument can be applied to the definition of `during` in [Ron97].

Counting is another operator where the detection time is not always clear. When the query is for "at least" n events, the complex event can be generated with reception of the n-th event. When the query is for "at most" or "exactly" n events, then the query has to wait for the end of the window that must be associated with the counting operator.

Some composition-operator-based languages additionally introduce a detection mode such as "immediate" or "deferred" [AE04, BZBW95]. While the former mode detects complex events as soon as possible, the latter has an additional lifetime or transaction that is associated with the query and the complex event will only be detected at the end of that. Note that such additional lifetimes can be considered unnecessary, because waiting until the end of the lifetime could also be expressed by placing the query as first operand in sequence and using an event signaling the end of the lifetime as second operand. The detection modes can be understood historically though, considering that in Active Database Systems, Event-Condition-Action rules (that could have complex event queries in the event-part) would often implicitly run inside database transactions.

3.2.6 Event Data and Correlation

So far, we have considered the types of events in complex event queries but not that data that events carry. Especially in the early literature on composition-operator-based languages, data has been treated with some neglect; these works would explain composition operators but make little or no reference to how event data affects event composition.

It is a common requirement to compose only events that "belong" together by being correlated on some data attributes. Consider an incoming event stream that has primitive events of types "order," "shipping," and "delivery." A complex event "order completed" could be detected as events of these types occurring sequentially. However, the incoming event stream contains all order, shipping, and delivery events that are produced, not just those for a given business transaction. The complex event query therefore must ensure that an order event is only composed with shipping and delivery events that are part of the same business transaction. To this end, events might have a data attribute "order number" and the query would have to only compose events with the *same* order number.

It would be a possibility to split the event stream with a mechanism separate from the complex event query, so that we have separate event streams per business transaction and each event stream then automatically contains only events that "belong" together by having the same order number. In Active Database Systems, such an implicit correlation of events was often the case when an event stream would only contain events from the same database transaction. This explains, in part, why many early languages neglected data in the event composition.

However, such an implicit event correlation cannot be assumed in the general case. Many complex event query compose events from different business or database transaction. An example would be a complex event query that involves counting the number of orders per day. Further, correlation is not necessarily on a single attribute. Instead of the order, shipping, and deliver event all having a single attribute "order number," the order event might have an "order number," the

shipping event an "order number" and a "tracking number," and the delivery event only a "tracking number."

Making the core composition operators sensitive to data in events is fairly straightforward, when the data in events consists of simple attribute-value pairs. With O, S, D being the respective event types for order, shipping, and delivery, the above example would then conceptually be $O(x); S(x, y); D(y)$, where the variable x is bound the order number and y to the tracking number.

The original event composition operators of XChange [Eck05, BEP06b, BEP06a] and the TREPL language [MZ97] are examples of languages that support such data-sensitive operators. XChange can compose events that are represented as XML messages and therefore have data with a more complicated structure than simple attribute-value pairs. In Amit [AE04], events composition is sensitive to so-called local and global keys. This can be deemed less expressive and flexible than arbitrary variables since there can only be a single key per event query (e.g., the correlation over order number and tracking number is not possible with a single key).

Counting operators are more difficult to extend to be sensitive to data. This is largely due to the fact that data can heavily affect what events are counted. One query might simply count the number of orders in a given time window, while another might count the number of orders with distinct items. Counting can also require some correlation with events that are not counted, for example a query might count the number of inquires received for a particular order (requiring a correlation on the order number). Most languages so far do not cater for these variation, however, and propose only a single counting operator. The language in [ME01] offers two counting operators `timesEq` and `timesDf` to count only events with the same values or only events with different values.

3.2.7 Event Instance Consumption and Selection

The same primitive event can be used multiple times to generate different complex events for a given query and several complex events can be generated at the same time. Consider the event query $A; B$ against an event stream a_1, a_2, b_1, b_2 (a_1, a_2 of type A, b_1, b_2 of type B). The event a_1 will be used twice to generate two different complex events, one when b_1 is received and one when b_2 is received. When event b_1 is received, two different complex event are generated at the same time, one for a_1 and one for a_2.

Some languages offer ways to restrict the reuse of primitive events and the generation of simultaneous complex events. Originally, this issue has been introduced in [CKAK94], where so-called parameter contexts were applied to event queries. Five different contexts were supported: unrestricted, recent, chronicle, continuous, and cumulative. Later this idea has been refined and the issues separated into event instance consumption, which restricts the reuse of primitive events, and event instance selection, which limits the generation of simultaneous events by considering only selected primitive events for composition [ZU99].

Consumption allows to invalidate certain events so that, once they have been used in one answer to a query, they cannot be used in other, later answers [GD94, ZU99]. Consumption can be relative to single query, i.e., reuse of the consumed event is not possible for this single query, or to a set of queries, i.e., also other queries will not reuse a consumed event. Note that consumption has direct impact on semantics and should be distinguished from event deletion as a garbage collection mechanism in evaluation algorithms (cf. Chapter 15).

Event instance selection allows to restrict the instances of an input event type that are considered for a composition with other input events so that, e.g., all event instances are considered, or only the first, n-th, or last instance is considered [ZU99, AE04, HV02]. Note that notions such as first or last only have an intuitive meaning when there is a linear order on the occurrence times of events; however usually there is no clear linear order when events happen over time intervals not points.

In general, both instance selection and event consumption should be sensitive to event data (e.g., for each order number, select the last inquiry event), an aspect that has usually been ignored in the literature so far. An exception is instance selection in Amit, which is sensitive local or global keys [AE04].

3.2.8 Formal Semantics and Representation of Answers

Although many composition-operator-based languages have been defined, these often lack precise and formal semantics. A particular question raised is how an answer to a query, i.e., a complex event, is represented both as a mathematical object and as a data structure that is provided for the execution of reactions to the complex event.

Some works try to give semantics by defining a predicate $O(q, t)$ which is true when there is an answer to the query q with occurrence time t [GA02, AC06] (similar in [CKAK94]). The time interval-based sequence operator would have the following definition for O then [GA02]:

$$O(\ (E_1; E_2),\ [t_1, t_2]) \iff \exists t \exists t'.\ t_1 \leq t < t' \leq t_2 \wedge O(E_1, [t_1, t]) \wedge O(E_2, [t', t_2])$$

It is worth pointing out that the sequence operator has one of the shortest definitions. A difficulty of this approach is that it only accounts for the occurrence time, but not the fact that at the same time several complex events may be generated (cf. examples earlier in Section 3.2.7). The reason for this is that complex events (answers to queries) do not have any explicit representation. Further this approach does not account for event instance consumption and selection and data in events.

An alternative approach is to use an explicit representation of complex events as a set or sequence of their constituent events, that is, of the primitive events used in answering the complex event query [HV02, CL04]. Every operator is then associated with a function that maps the event histories of its operands to complex events. The event histories of operands are the sets or sequences of events that have been produced so far for the operands; note that the elements can be complex events themselves (i.e., sets of primitive events). This approach can account for event instance selection. It does not account well for event instance consumption, however, because this essentially requires changing the histories of the operands whenever complex events are detected [HV02]. In its basic form, the approach usually does not account for data. However, a fairly similar approach in [Eck05, BEP06b] can cope with data as variable bindings.

3.3 Data Stream Query Languages

The second style of languages has been developed in the context of relational data stream management systems. Data stream management systems are targeted at situations where loading data into a traditional database management system would consume too much time. They are particularly targeted at near real-time applications where a reaction to the incoming data would already be useless after the time it takes to store it in a database. Having been developed in the database field, query languages for data streams put their focus more strongly on data than composition-operator-based languages. On the other hand, they pay less attention to issues such as timing and temporal relationships that are important when processing events.

Much of the research on data streams focuses on optimizing particular classes of queries (e.g., different forms of filtering, joins, aggregation). These queries are then only expressed in a SQL-like straw-man query language without much work on language design or formal language definition. An important exception of this is the Continuous Query Language (CQL) that is used in the STREAM systems [ABW06]. There are so far no significantly different competing language proposals in the data stream area. The general ideas behind CQL also apply to a number of open-source and commercial languages and systems including Esper [Esp], the CEP and CQL component of the Oracle Fusion Middleware [Ora], and Coral8 [MV07]. In the following, we base our introduction to data stream query languages therefore on CQL as described in [ABW06]. However, we also discuss ideas and extensions from other systems that share the same basic model.

3.3.1 General Idea

A data stream is an unbounded sequence of time-stamped tuples. The time-stamps are only time points (not intervals). In the context of complex event processing, an event typically corresponds

```
SELECT Istream O.customer, S.trackingId
FROM O[Range 2 Hours], S[Range 2 Hours]
WHERE O.id = S.orderId
```

Figure 3.1: A first example of a CQL query

to a single tuple in a data stream, and the data stream that is belongs to has some correspondence with an event type.

CQL aims at leveraging SQL for querying such data streams. However, SQL operates on relations (or tables), which are in contrast to streams finite and not time-stamped. Therefore SQL cannot directly operate on data streams.

The way that CQL applies SQL to data streams is essentially the following: For each point τ in time, the data streams received so far are converted into relations. Then the query is evaluated on these queries as a regular SQL query. The result can then be converted back into a stream. Note that CQL requires that time points are taken from a discrete domain (e.g., integers). Regarding semantics, it is important to emphasize that the conversion process from stream to relations and back to streams happens conceptually at each point in time. The actual evaluation is however for many queries more efficient by avoiding the conversion to relations and operating directly on streams.

For converting streams to relations, CQL supports a number of so-called *stream-to-relation* operators. An example of such a stream-to-relation operator is a sliding time window (of length d) that produces a relation containing all tuples with a time stamp between the current time τ and $\tau - d$ from a stream.

Transformations between relations (*relation-to-relation* operators) are performed using standard SQL queries, often with some restrictions (e.g., no nested queries). The abstract semantics of CQL would allow to use other relational query languages as well, but this has not been done in practice so far.

For converting relations back to streams, CQL supports a number of *relation-to-stream* operators. An example of such relation-to-stream operator is the `Istream` operator, which produces a stream tuple for each tuple that has been inserted into the relation at the current time τ (in comparison to the previous state of the relation at time $\tau - 1$).

Note that in the original proposal of CQL [ABW06], there are no stream-to-stream operators. However, recently such operators, which are somewhat similar to the composition-operators discussed earlier, have been proposed. These will be discussed later and separately in Section 3.6.1.

As a first example of a CQL query consider that a stream `O` contains order events (for some `customer` and with a number `id`) and a stream `S` contains events that signify that an order (with number `orderId`) has been shipped (with tracking number `trackingId`. The CQL query in Figure 3.1 pairs up events with the same order number over a time span of two hours and produces as output a stream containing customers together with the tracking numbers of their orders. Note that since the relation-to-stream operator is `Istream`, there will be only one output tuple per pair of order and shipping events. The output tuple will carry the time-stamp of the later of the two events.[7]

We now look at the different stream-to-relation and relation-to-stream operators in more detail. We then get back at the full picture of CQL by looking at how these operators are used together with SQL as relation-to-relation operator to write common types of event queries. Finally we will discuss semantics of CQL together with issues related to the detection time of events.

3.3.2 Stream-to-Relation Operators

The task of a stream-to-relation operator is to produce a relation $R(\tau)$ at a time point τ from the tuples received up until τ in stream S. Typically the relation is simply a selection of tuples from

[7] Intuitively, a shipping event should always come after its order event. However, the query does not prescribe this and would also allow order and shipping events to arrive in different order.

the stream based, e.g., on the time stamp of the tuples. Since the relation contains only a finite extract of the stream, one also refers to a stream-to-relation operator as a *window* that is applied to the stream.

The syntax for applying a stream-to-relation operators to a stream `S` is `S[`*spec*`]`. The resulting relation may then be used in the `FROM`-clause of a SQL expression. The window to select tuples from the stream is specified in *spec*.

The most intuitive forms of windows select tuples based on their time stamps. Common forms of time-stamp-based windows found in data stream management systems include:

- Now window: The resulting relation contains only the stream tuples with the time stamp τ. The syntax for this window is `S[Now]`. In the earlier example of order and shipping events, the query could be modified to use `S[Now]` instead of `S[Range 2 Hours]` if we assume (as would typically be the case) that shipping events arrive after their corresponding order events.

- Unbounded window: The resulting relation contains all stream tuples received up until the time τ. The syntax for this window is `S[Unbounded]`. Note that unbounded windows often —but, as we will see, not always— lead a forever increasing demand in memory over time since the resulting relations grow with every new tuple in the stream. The syntax for unbounded windows is `S[Range Unbounded]`.

- (Simple) sliding windows: The resulting relation contains all stream tuples between $\tau - d$ and τ, where d is a specified duration such as "1 minute" or "2 hours." The syntax for a sliding window of duration d is `S[Range` *d*`]`. For example a 30 second window would be written as `S[Range 30 Seconds]`.

- Sliding windows moving at fixed granularity: The resulting relation also contains tuples within a window of a specified size d. However, the window does not move continually but over a grid of s time units. In other words, every s time units, the start of the time window is moved forward by s time units. The syntax for a window of duration d that moves by a granularity of s is `S[Range` *d* `Slide` *s*`]`. For example a 24 hour window that moves in 1 hour increments would be written as `S[Range 24 Hours Slide 1 Hour]`

- Tumbling windows: These are a special case of sliding windows with fixed granularity, where the window duration d is the same as the sliding granularity s, i.e., $s = d$. Note that the windows do not overlap in this case and the end of one window becomes the start of the next.

- User defined: The resulting relation contains tuples within time points $f(\tau)$ and $g(\tau)$, where f and g are user-specified functions. Typically f and g are functions external from the data stream management system, programmed, e.g., in Java or C++. It is required that $f(\tau) \leq g(\tau)$ for all τ.

Instead of selecting tuples based on their time stamps, windows can also have a fixed capacity of tuples. At time point τ, the tuple-based window specification `S[Rows` *N*`]` would contain the N most recent tuples of the stream `S`, that is, those with the largest timestamps smaller or equal τ. If there are several tuples with the same time stamp then it can be ambiguous, which of the older tuples are in the relation in which are not. Tuple-based windows are computationally preferable to time-stamp-based windows, since the size they take up in memory is fixed and known in advance. Tuple-based windows can be useful in applications that average over measurements such as stock ticks or sensor values in order to smooth and clean data. However, time-based windows are often more natural in event processing both since they are easier to understand for human users and because many business applications (e.g., order processing) refer directly to time.

A variant of tuple-based windows are partitioned windows. Partitioned windows split the stream into substreams based on the values of some attribute (similar to a `GROUP BY` clause in SQL). Then a tuple-based window is applied separately to each substream. The result relation

then is the union of all the relations for the substreams. Note that partitioned windows usually do not have a fixed size since the number of substreams depends on the (varying) number of different attribute values. The syntax for partitioned windows is `S[Partition By` $A_1, ...A_k$ `Rows` N`]`, where $A_1, ...A_k$ are attributes of `S`.

Windows based on time-stamps or tuple order always contain recent tuples and tuples enter and leave the window in a first-in-first-out (or first-in-first-expire) manner. Predicate windows [GAE06] break this pattern and offer a more flexible approach, where tuples qualify and disqualify for the window based on a predicate. The predicate can be defined over any attribute of the stream. Note that the predicate windows in [GAE06] rely on tuples having a so-called correlation attribute. A tuple with the same value for the correlation attribute as a previous tuple is considered an update to that tuple and overrides the previous tuple.

Similar to predicate windows are the so-called basket expressions of the DataCell system introduced in [KLG07]. In the DataCell, tuples arriving in streams feed into so-called baskets. The basket produced from a stream is essentially the same as the relation produced by the unbounded window in CQL. Tuples can be removed from baskets, however, using so-called basket expressions. Basket expressions are regular SQL queries surrounded by square brackets. In contrast to SQL queries they have a side-effect: they consume tuples, i.e., input tuples that are used in generating an output tuple are removed from their baskets. Note that this is reminiscent of consumption modes in composition-operator-based languages (cf. Section 3.2.7). Because predicate windows decide what tuples *remain* in a window and basket expressions decide what tuples are *removed* from a basket, the two can be understood as being loosely symmetrical. However, offering a more fine-grained control, basket expressions are more flexible. Since they rely on side-effects, the DataCell also requires a more involved computational model based on Petri-Nets and thus does not use the stream-relation conversion model and semantics of CQL. Aggregates and joins must also be programmed on a more fine-grained level in the DataCell.

3.3.3 Relation-to-Stream Operators

As explained earlier, regular SQL queries are applied to the relations produced by the stream-to-relation operators. The result of these queries is a relation $R(\tau)$ for each time point τ. In order to get a stream of time-stamped tuples as a result rather than relations, relation-to-stream operators are applied. There are three relation-to-stream operators in CQL: `Istream`, `Dstream`, and `Rstream`.[8]

`Rstream` stands for "relation stream". This operator produces a stream from its input relations by simply inserting all tuples of $R(\tau)$ into its output stream with a timestamp of τ. Note that since $R(\tau)$ often differs only slightly from its previous state $R(\tau-1)$, the same tuple will often be found several times in the output stream with different time stamps.

In contrast, `Istream` and `Dstream` do not produce the full state of a relation for each time point τ, but only the changes to the previous state. `Istream` stands for "insert stream" and it contains the tuples that are inserted into the relation R. More precisely, it contains a tuple t with time stamp τ whenever t has been added to $R(\tau)$ in comparison to the previous state of R, i.e., $t \in R(\tau) - R(\tau-1)$. `Dstream` stands for "delete stream" and is the opposite containing the tuples that are deleted from the relation R. More precisely, it contains a tuple t with time stamp τ whenever t has been removed from $R(\tau)$ in comparison to the previous state of R, i.e., $t \in R(\tau-1) - R(\tau)$.

Note that these relation-to-stream operators rely on a discrete notion of time, since the definitions of `Istream` and `Dstream` make use of a notion of the previous state of a relation identified by $R(\tau-1)$. If the time domain was dense not discrete (e.g., isomorphic to the rational number not the natural numbers), then the previous state of the relation cannot be identified by a time point $\tau-1$.

[8] Note that in the Esper language [Esp], the `Rstream` operator corresponds to the `Dstream` of CQL and there is no operator with the effect of the `Rstream` operator of CQL.

```
SELECT Istream(*)
FROM O[Range Unbounded]
WHERE total > 1000
```

(a) Istream, Unbounded

```
SELECT *
FROM O
WHERE total > 1000
```

(b) Syntactic shortcuts

```
SELECT Rstream(*)
FROM O[Now]
WHERE total > 1000
```

(c) Rstream, Now

Figure 3.2: Simple filter queries for “big” orders in CQL

3.3.4 SQL as Relation-to-Relation Operator

The abstract model of CQL semantics would in principle allow for any kind of relation-to-relation operation. However in practice, relation-to-relation operators are always expressed in SQL (or a somewhat restricted subset of it). We now look at some common queries in CQL using the stream-to-relation and relation-to-stream operators discussed so far and SQL as a relation-to-relation operator.

In addition to the pure query language as will be discussed here, languages in data stream systems also contain constructs for defining schemas, creating and removing streams, indexes, etc.; these construct are similar to their corresponding SQL constructs (e.g., `CREATE TABLE`, `DROP TABLE`) and of little relevance here.

Simple Filters Filters, which select tuples from a stream based on attributes, are the simplest kind of query. They cannot really be considered complex event queries, since they apply to single tuples or events without combining information of several tuples or events. However they are still interesting to illustrate some salient points about CQL.

Consider again that `O` is a stream of order events. The query in Figure 3.2(a) filters this stream so that the output contains only “big” orders, i.e., orders with a total of more than $1 000. For each time point τ, the `Unbounded` operator produces a relation containing all orders with a time stamp earlier or equal τ. This relation is filtered to contain only big orders (WHERE clause). The `Istream` operator then converts the relations back into a stream. Since `Istream` produces only those tuples that are added to the relation, each big order in the input stream `O` is also only contained once in the output stream.

For a simple filtering query like this, the round trip from stream to relation, to relation, and back to stream, might be considered somewhat unintuitive. In particular, although the window specified in the query is unbounded, the query in fact does not have unlimited memory demands since it can be evaluated on a per-tuple basis without storing tuples of `O` at all.

This weakness in the language design of CQL is somewhat remedied by syntactic defaults that allow to write the query in a more intuitive form without the unbounded window and the `Istream` operator as in Figure 3.2(b). Note however, that this is just a syntactic shortcut and its expanded form is still that of Figure 3.2(a). The query can also be written in a different way using the now window and an `Rstream` operator as in Figure 3.2(c).

Joins Tuples from different streams can be joined. Unlike the tuples produced by simple filtering queries as above, output events (or tuples) of queries involving joins are generated from several input events (or tuples) and thus are “real” complex events. An example of a query involving a join as already been given earlier in Figure 3.1. Joins between data streams correspond the conjunction operator in composition-operator-based languages (Section 3.2.3).

The streams that are joined should always have a finite window; an unbounded window on one of the streams would lead to unbounded memory requirements for storing that stream. Note that determining the appropriate windows for streams can be a difficult task for some event queries. As an example consider having to join order, shipping, and delivery events. The events happen in this sequence, order and shipping within 24 hours, and shipping and delivery within 48 hours. The correct windows then would be sliding windows of a duration of respectively 72 hours and 48 hours for the order and shipping event streams, and a now window for the delivery event stream.

```
SELECT Istream( count(id) )
FROM O[Range 24 Hours]
WHERE O.total > 1000
```

(a) Counting big orders

```
SELECT Istream( payment , count(id) )
FROM O[Range 24 Hours]
GROUP BY O.payment
```

(b) Counting orders per payment type

Figure 3.3: Aggregates in CQL

This issue will resurface in a different shape in connection with the temporal relevance and garbage collection in XChange^{EQ} in Chapter 15.

Joins in CQL usually make no constraints to the order in which tuples from different streams arrive, i.e., there is no real equivalent of the sequence operator in composition-operator-based languages (Section 3.2.2). In general it can be said that beyond time-stamp-based windows, CQL offers little support for such temporal aspects of event queries. On the other hand, CQL has a strong emphasis on data and supports more flexible correlation between events than most composition-operator-based languages. For example the "order completed" query from Section 3.2.6, which joins order and shipping events based on an order number and then shipping and delivery events based on a tracking number, can be expressed easily in CQL.

Aggregates By using SQL's aggregation functions such as `MAX`, `MIN`, `COUNT`, `SUM`, or `AVG`, data from multiple tuples can be aggregated. Consider the query in Figure 3.3(a), where `O` is again a stream of order events. The query counts how many "big" orders (total > $1 000) have been places within the last 24 hours. Since the relation-to-stream operator is `Istream`, output is generated whenever the number changes by either a new big order coming in or an old big order falling out of the 24 hour window. Note that if the `Istream` operator is replace by `Rstream`, then the query will generate an output tuple at every time instant, whether the value changes or not.[9]

By using the `GROUP BY` clause of SQL, tuples can also be first grouped together and then aggregated on a per group basis. The query in Figure 3.3(b) counts the number of orders separately for each type of payment (e.g., credit card, check). By applying an additional `HAVING` clause, one could filter the output further, e.g., to generate output tuples only when the count exceeds a given number.

In comparison to composition-operator-based languages, aggregation of data is much better supported in data stream languages, especially from a single stream. Whether a given aggregation query can be expressed in a data stream language depends crucially on the supported windows. For example, a data stream language without support for tuple-based windows would have difficulties to express a query such as "compute the average over the last 10 sensor values." Further, most current data stream management systems only support time-based and tuple-based windows. When the window over which data is to be aggregated is however based on other events (e.g., "compute the number of failed requests between the server being switched off and on again"), this becomes hard to express.

There has been a significant amount of work on aggregation queries on data streams that have a high or bursty throughput. Techniques such as load shedding allow data stream management systems to maintain their performance (in terms of throughput and latency) at the cost of precision, e.g., by computing only approximate aggregates. There is a considerable number of applications where such imprecision is tolerable, e.g., processing of sensor data, where already incoming data is subject to measurement imprecision, but also some computations on stock market data which are ultimately interested in trends (rather than some precise value).

[9] Note that some data stream management systems would interpret time as advancing only whenever a new tuple comes in or an old tuple leaves its window. In this case, the query would return the almost the same result for `Istream` and `Rstream`. The only difference then is in thouse cases when an equal number of tuples enters and leaves the window at the same time where only the `Rstream` operator will generate output.

```
SELECT Rstream( O.id, C.price, C.productGroup )
FROM O[Now], C
WHERE O.item = C.item
```

Figure 3.4: Access to database tables in CQL

Negation Negation (in the sense of absence of events or tuples) is less well supported in data streams. The reason for this is that negation queries in event processing typically require a window that is determined by other events, e.g., "event A does not happen between event B and C" or "event A does not happen within 1 hour of B." Thus, while the time-based and tuple-based windows of data stream languages serve well for most common aggregation queries, they are often insufficient for negation queries, even though negation and aggregation are otherwise quite similar.

In those cases where an appropriate window can be specified, negation must be expressed as in SQL using either using a count aggregate (`COUNT(...)=0`) or the `NOT IN` set operation.

Views Most data stream query languages support the definition views over streams. Like views on tables in SQL, a view on streams is defined by assigning a name to the result stream of a specified query. This result stream can then be used as input in other queries by referring to its name. Views are an important construct for organizing large query programs into smaller units that logically belong together. Stream views often do not allow recursive cycles, i.e., in the definition of a stream view one cannot refer directly or indirectly the this stream view. Computations such as transitive hulls that require such recursive cycles thus cannot be expressed with these views.

Access to Database Tables Streaming tuples can also be joined with persistent relations. Particularly common are queries that "enrich" events, i.e., add data to events using information stored in a persistent database. For example a stream of order events might only contain a number of the ordered item, however some query might be interested in the price and the product group (e.g., books, electronics) of the ordered item. Then it is necessary to obtain this information from a database table that contains among other information the price and product group for each item number. Figure 3.4 shows how a query realizing this would be expressed in CQL. `O` is again a stream containing order and `C` is a database table containing information about items, in particular the product group of an item.

It should be noted that such joins with persistent relations are most times only meaningful when using a now window as stream-to-relation operator and the `Rstream` stream-to-relation operator. Otherwise a query might produce new output tuples at unexpected times, e.g., when the price in the database table changes. Although it is generally a strong point of data stream query languages that they allow such combined access to stream and non-stream (database) data, the issues when database data changes cannot be considered fully solved.

3.3.5 Semantics and Detection Time

The semantics of CQL rely on a discrete notion of time: At each point τ in time, the streams received so far are converted into relations, a query is evaluated on these relations to yield a result relation, and this result relation is then converted back into a stream. Importantly, the conversion back into a stream requires knowledge not only of the current result but also the result at the preceding time point $\tau - 1$ in the case of the `Istream` and `Dstream` operator. Further, the `Rstream` operator outputs the full result relation at *every* time point. Unless the query is to a single stream to which the now window is applied (so that the resulting relation is empty when no new events have been received), the `Rstream` operator also requires a discrete notion of time.

A dense time model (e.g., isomorphic to rational or real numbers) might in general be perceived more natural that a discrete time model (e.g., isomorphic to integers). This is true even when the actual realization in a computer system will always be discrete due to limited precision in the

representation of numbers; the granularity imposed by this precision might be too fine for a given application. For example, when the `Rstream` operator is used, it is for performance reasons not desirable to output a relation, e.g., every millisecond when an application is content with every second. Specifying a fixed granularity of the time model that meets the needs of an application would a conceivable approach, however this means that there is some external mechanism that impacts the semantics and results of queries.

Rather than outputting the full relation at every time instant with the `Rstream` operator, it might be more desirable to output the full relation only whenever a new tuple is received.[10] It would be possible to use an artificial "discretization" of regular wall-clock time so that a logical clock advances on step whenever a new event is received. However there must then be a mapping between this logical time and the wall-clock time to properly deal with time-based windows. Further, the logical time must advance not only when new events are received, but also when old events or tuples leave their time-window.

The semantics of CQL rely on the trinity of stream-to-relation, relation-to-relation, and relation-to-stream operators. Understanding a query thus always involves a full round trip from streams to relations, to a result relation, and then back to a stream. In order to know the result at a given time point τ, it is not sufficient to apply this process only for the time point τ when an `Istream` or `Dstream` operator is involved. The reason for this is that these operators deliver a difference between the current state of the result relation and the previous state of the result relation. Fully understanding a query might thus be a tedious task that involves mentally replaying the full streams for each time instant, especially for large query programs that use views over streams.

CQL and other data stream languages offer only little support for queries that involve temporal relationships between events or tuples and do not support events occurring over time-intervals. In CQL, time is primarily referred to with time-based windows. On a secondary level, queries might refer to time in the SQL (relation-to-relation) part of queries as regular data attributes of tuples.

In summary, CQL has very precise semantics for data but these semantics can be considered somewhat unintuitive and less precise concerning detection time of complex events. Firstly, semantics always involve a round-trip from streams to relations, to a result relation, and then back to a stream. Secondly, they rely on a discrete time domain, which has some difficulties. In contrast, composition-operator-based languages seem very intuitive in their semantics and offer good support for temporal relationships, but have several hidden problems of imprecision (e.g., semantics that do not account for data in event, cf. Section 3.2.6) and potential misunderstandings (e.g., variations on the interpretation of the sequence operator, cf. Section 3.2.2).

3.4 Production Rule Languages

Production rules are not an event query language as such, however they offer a fairly convenient and very flexible way of implementing event queries. The primary reason that production rules are more convenient for implementing event queries than a general purpose programming language is that their forward-chaining evaluation algorithm is very similar to the algorithms used for evaluating event queries (see Chapter 12). In addition, many business rules management systems (BRMSs) that are based on production rules offer support for defining domain-specific languages (DSLs). A domain-specific languages hides the syntax of production rule languages as well as some intricacies of the object model behind an interface of controlled natural language.

The first successful production rule engine has been OPS [FM77], in particular in the incarnation OPS5 [For81]. Since then, many others have been developed in research and industry, including Drools (also called JBoss Rules) [JBo], ILOG JRules [ILO], and Jess [San]. The examples in this section are in the Drools Rule Language (DRL), but other production rule languages

[10] Note that many data stream management systems were perceived for processing high throughput (e.g., stock tick data from financial markets), where there might be the implicit assumption that there is always at least one tuple per time instant being received. In these cases there would be no difference. However, when data stream languages are applied in other contexts, this assumption might not be sensible.

are very similar. While the general ideas of production rules will be explained here, we refer to [BBB+07] for a deeper introduction.

A production rule, sometimes also called condition-action (CA) rule (in contrast to event condition action rules), is a statement of the form `WHEN` *condition* `THEN` *action*. It specifies that the action is to be executed whenever the condition becomes true. It should be emphasized that the action is only executed once when the condition *becomes* true (i.e., whenever its truth value changes from false to true), not every time the condition holds (i.e., has the value true).

The condition is evaluated over a set of facts called the working memory. A production rule engine is usually tightly coupled to a general purpose programming language such as Lisp or Java. Accordingly, facts are typically represented in the data model of that host programming language, i.e., as Lisp terms or Java objects. Facts must be explicitly inserted into or deleted from the working memory. The operations a production rule engine offers for this are usually called *assert*[11] and *retract*.

There are no restrictions on the action, it can be an arbitrary method or procedure call in the host programming language. Particularly interesting are actions that change the working memory by asserting new facts, retracting existing facts, or updating existing facts. Production rules can be used to implement deductive inferences by simply asserting the deduced facts. To this end, many production rule languages also offer an *assert logical* action (in addition to the standard assert), which has the effect that the asserted fact is automatically retracted when its associated condition becomes false again.

Production rules are evaluated in a forward-chaining manner in so-called match-act cycles. Their evaluation must be explicitly invoked with a method call such as `fireAll()` to the production rule engine from the host programming.[12] When invoked, the production rule engine checks all rule conditions against the working memory ("match"). From all rule instances that can fire, it selects a single one according to some conflict resolution strategy. The action of this single rule instance is then executed ("act") and possibly modifies the working memory. This match-act cycle is then repeated until there are no rule instances that can fire. (Note that non-terminating rule sets are possible.) Algorithms such as rete [For82] can realize the matching in a fairly efficient manner by incrementally updating the results of the match phase when the working memory changes and thus avoiding to re-check every condition in every match-act cycle.

3.4.1 General Idea

Production rules can be used to implement event queries "manually" by (1) asserting some fact for each event that happens and (2) writing each event query as a condition over these facts. To ensure timely detection of complex events, the evaluation of the production rules must be invoked (with `fireAll()`) after each assertion of an event fact. Using production rules for CEP essentially means that a complex event query is transformed into an expression over the state of the working memory.

Figure 3.5 shows how the complex event that an order has been completed (again, this consisting of order, shipping, and delivery) might be realized with a production rule in the Java-based Drools Rule Language (DRL). The rule assumes that Java objects of classes `Order`, `Delivery`, and `Shipping` are asserted in the working memory whenever respective events happen. The classes have attributes for the order number (`Order.id` and `Shipping.orderId` and the tracking number (`Shipping.trackingId` and `Delivery.id`). The rule fires and performs its action (simple console output), whenever (1) a combination of objects o, s, d of the three respective classes can be found in the working memory that satisfies the conditions on the order number and tracking number and (2) the rule has not already been fired before for this particular combination.

[11] Since many newer programming languages use "assert" as a keyword, the assert operation is now also often renamed to "insert" to avoid clashes.

[12] More precisely, the method call is usually not directly to the production rule engine object, but rather a so-called session object which bundles an instance of a working memory together with one or more rule sets.

```
rule "Detect completed orders"
  when
    o: Order(),
    s: Shipping(),
    d: Delivery(),
    o.id == s.orderId,
    s.trackingId == d.id
  then
    System.out.println("Order " + o.id + " completed.");
end
```

Figure 3.5: A production rule for detecting "order completed" complex events

```
rule "Detect completed orders"
  when
    o: Order(),
    s: Shipping(),
    d: Delivery(),
    o.id == s.orderId,
    s.trackingId == d.id,
    Helper.withinHours(o.timestamp, s.timestamp, 24),
    Helper.withinHours(s.timestamp, d.timestamp, 48),
  then
    System.out.println("Order " + o.id + " completed on time.");
end
```

(a) Production Rule

```
import java.util.Calendar;

public static class Helper {
    public static boolean withinHours(Calendar t1, Calendar t2, int d) {
        Calendar t1PlusD = t1.clone().add(Calendar.HOUR, 24);
        return t2.before(t1PlusD);
    }
}
```

(b) Auxiliary Java class

Figure 3.6: Production rules and temporal conditions

3.4.2 Temporal Aspects

Production rules are designed to operate on facts, not events. Therefore, production rule languages typically offer no designated construct for expressing the temporal aspects commonly needed in event queries. However, it is possible to some degree to implement such temporal aspects of an event query as a condition on attributes of event object.

For example, each event object could carry an attribute `timestamp` that signifies the occurrence time of the event. Whether this is a time point or a time interval is a design choice left to the programmer. Because production rules rely on the types and functions that are available in their host programming language and because time points are better supported in most programming languages, using time points might often be convenient where they are sufficient.

Consider a refinement of the "order completed" complex event to a "order completed on time" complex event. This event has as additional constraint that an order must be shipped within 24 hours, and the shipping delivered within 48 hours. When the respective classes for the order, shipping, and delivery event have an additional attribute `timestamp` (e.g., of the Java type `Calendar`, which represents time points), then these temporal constraints can be expressed as conditions on these attributes. Figure 3.6(a) shows a modified version of the earlier "order completed" rule from Figure 3.5, where the time constraints for "order completed on time" have been added. This rules makes use of an external helper function `withinHours` that realizes the necessary temporal computations on the Java `Calendar` objects. A possible Java implementation of this helper function

is shown in Figure 3.6(b).

While it is possible to express temporal conditions as conditions on attributes of events, production rules offer no guidance to programmers on how to express such temporal conditions and how to design their event classes appropriately (e.g., with attributes for time stamps). As will become evident later, the way the example rules given here are written is influenced significantly by the design of XChange^{EQ}. Neither production rule languages nor Java offer sufficient support for typical temporal conditions of event processing. Helper functions such as the one in Figure 3.6(b) will therefore be needed quite often. Note that such helper functions can have arbitrary side-effects. In the example it would be easy to forget calling `clone()` and thus mistakenly *modify* a time stamp of an object in the working memory in the helper method. Finally, the evaluation of production rules is unaware of the special semantics of temporal conditions and cannot use them to optimize query evaluation.

Production rules also offer little support for temporal events. If temporal events, be they relative (e.g., "12 hours after event X") or absolute (e.g., "at noon"), are needed in an event query, then a fact must be asserted at the time this temporal event is supposed to happen. This must be done externally from the production rule engine in regular Java code. This in turn entails the use of fairly low-level functionality to schedule the execution of a thread that asserts the corresponding fact for execution an a given time (e.g., using Java's `Timer` class) and a potential for race conditions. Also, it violates the principle that the event processing logic should be encapsulated in the production rules: it can then not be simply changed by changing the production rules, other (external) code must be changed as well.

Some production rule languages allow to delay the execution of a rule by a given duration (in Drools this is expressed the `duration` keyword). This might be used in some cases as a substitute for relative temporal events, but is fairly limited.

3.4.3 Garbage Collection

Asserting a fact for each event that happens raises a practical concern: over time, the number of facts and thus the size of the working memory grows and grows. Since we often assume unbounded streams of events in event processing, we will eventually run out of memory if we only assert facts and never retract them. To avoid this we need some way to retract event facts that have become irrelevant, i.e., some kind of garbage collection of event facts.

Since current production languages are not designed to work with events and their temporal relations, they usually offer no help for garbage collecting event facts. Note also that the automatic garbage collection of the host programming language (e.g., Java) will not help: the working memory keeps a list of references to the objects representing its facts, so that any object in the working memory is accessible and will never be subjected to automatic garbage collection. Essentially, the programmer is left with programming garbage collection of event facts manually. This can be done either by writing further production rules that retract events that have become irrelevant or externally in the host programming language.

Garbage collection raises the question what it means for an event to be relevant, an issue that will also be the subject of Chapter 15 of this thesis. A simple method might be to define a default timeout for each event type. Any event older than the timeout is then deemed irrelevant. However, this definition influences the logic and semantics of event queries since different timeouts might lead to different query results.

In general, it would be preferable to use temporal (and other) conditions in the queries to figure out when an event becomes irrelevant. Consider again the "order completed on time" query. We can reason that a shipping event that is older than 48 hours is irrelevant to this particular query, since there will be no joining delivery event that satisfies the constraint that shipping and delivery must happen within 48 hours. However, for the order event things are more complicated. It might seem that the same reasoning can be applied to deduce that it becomes irrelevant after 24 hours. However this is not correct: if a joining shipping event has been found within those 24 hours, the event must be kept in the working memory to join order and shipping with a later delivery event. Therefore an order event might be relevant for up to 72 hours. Further, this reasoning only

concerns the single "order completed on time" query. There might be further event queries, e.g., a query to aggregate orders over a full week, that will lead to different relevance times.

We can see from this that figuring out whether an event fact is irrelevant or not —and thus can be garbage collected or not— can be hard. Therefore programming garbage collection manually is hard. Since an incorrect garbage collection will crucially affect the correctness of event queries and might lead to memory leaks, manual garbage collection of events is also somewhat dangerous. Further it leads to code that is hard to change and maintain: adding a single production rule for a new event query might affect the relevance of the event facts it accesses and thus entails adapting the code for garbage collection.

There has been some research on extending production rule engines and languages for event processing [Ber02, WBG08]. In particular, production rules are extended in [WBG08] by adding event composition operators and an automatic garbage collection of events. This work will be discussed in Section 3.6.3. Such approaches have not yet found their way into current production rule engines; however one can expect to see more work on this in the near future.

3.4.4 Negation and Aggregation

Negation and aggregation are supported in many production rule languages. For negation, the `not` construct makes it possible to test for the absence of certain facts in the working memory. For aggregation, facts can be collected into lists (using a list type of the host programming language such as `List` in Java) with a construct such as `collect` or the more general `accumulate`. The action of a rule can then use this list to compute an aggregate such as the average over an attribute. Note that this aggregate must usually be programmed manually in an iteration over the list. (A simple count aggregate is an exception, because lists typically provide a function giving their length.) In particular, the iteration must also take care of grouping and duplicate elimination if these are part of the event query specification.

The constructs for negation and aggregation are of course intended for processing normal facts rather than events happening over time. Their typical use in event queries, e.g., aggregation over a sliding time window or negation (absence) of an event between two other events, are therefore hard to express. Again this requires the use of conditions on time stamp attributes and similar mechanisms.

State-based processing, which we look at next, can in some cases also be used for expressing event queries involving negation or aggregation in production rules. It thus provides an alterative for the constructs `not` and `accumulate`. When to use state-based processing or these constructs, is a design choice left to the programmer and often not an easy one.

3.4.5 State-based Processing

A strength of production rules is that they allow to combine state-based information with event-based information. This is not surprising: Production rules are after all primarily intended for state-based processing, i.e., performing specific actions when the working memory enters specific states. When using production rules for event processing, asserting a fact for an event that happens can be understood as a state where that event has happened.

Some situations which might be perceived as a complex event at first glance are easier to specify and detect as a certain state rather than through a complex event query. Consider a room that is equipped with sensors signaling events `enter` and `leave` whenever a person enters or leaves the room through its door. An application for climate control might require to detect situations where more than three persons are in the room. Expressing this situation as a complex event, i.e., a combination of events happening over time, is typically hard: it is not simply a sequence of three `enter` event, because (1) we must know how many persons are in the room when the complex event query is started and (2) usually there will be `leave` events that cancel the effect a preceding `enter` event.

Using production rules and state-based processing, however, this situation if fairly easy to detect. We simply maintain a single fact to record the number of persons in the room. This fact

just has a single counter attribute of type integer. When `enter` and `leave` events happen, we will not assert facts for these events in the working memory. Rather we will only increase or decrease the counter. A production rule for detecting the situation where more than three persons are in the room then is simple to write (here in pseudo code): `WHEN` *couter* > 3 `THEN` *action*. Note that it is fairly ease to incorporate new events that affect the number of persons in the room (e.g., persons entering through the window instead of the door) into this scenario.

In contrast, both composition-operator-based languages and data stream languages will have significant difficulties for detecting such a situation. (One possible solution might be to count `enter` and `leave` separately and then put a threshold on their difference. However this query requires unbounded windows for both event types and the threshold depends on the number of persons in the room at the beginning.) In their excuse however, the described situation should conceptually really be seen as a state of the room object rather than a complex event.

State-based processing can in some cases also be used to program negation and aggregation. Instead of collecting event facts in a list and then performing a computation of an aggregate using this list, it is possible in some case to maintain only the aggregate value as a single fact (similar to the counter in the previous example). This aggregate value must then be modified by each event that happens. For aggregates such as maximum, minimum, count, or sum, the modifications are straightforward. An average can also be expressed by a combination of sum and count. A difficulty however arises when the aggregate is to be computed of a sliding window, e.g., counting the number of events within the last hour. Then it is not sufficient to just increase the counter when a new event happens. One hour after the event has happened, the counter must also be decreased. The decreasing of the counter requires essentially a relative temporal event, and the issues associated with temporal events and production rules have already been discussed in Section 3.4.2.

3.5 Comparison

Having introduces the three prevalent styles used for querying complex events, composition operators, data stream queries, and production rules, we now compare their strengths and weaknesses.

3.5.1 Support for Specific Query Features

The first aspect for comparison is how well each approach is suited for expressing certain types or features found in event queries.

Temporal Aspects Composition-operator-based languages offer a very strong support for event queries involving temporal events or temporal relationships between events. Data stream query languages offer less support for such temporal aspects; their main capabilities in this respect derive from temporal windows, which are fairly limited and inconvenient. Production rules usually offer no built-in support for such temporal aspects. However, it is possible to treat timestamps like regular (data) attributes and write appropriate functions for expressing temporal relationships. Generating temporal events is also possible, but hard and in violation of the encapsulation principle since it must be done in the host programming language outside of the production rules.

Negation and Aggregation Composition-operator-based languages offer good support for typical forms of negation of events. They offer little support for aggregation, in part because event data is an aspect that is often somewhat neglected in these languages. Data stream query languages excel at aggregation of event data. They are less well-suited for queries involving negation. In production rules, both negation and aggregation require a fair amount of manual, low-level implementation.

Consumption and Selection of Events Many composition-operator-based languages offer ways to consume and select event instances in order to limit their use and re-use in event queries. Prevalent data stream query languages offer no such mechanism. (Basket expressions discussed in Section 3.3.2 are an exception.) In production rules, consumption and selection can be programmed manually, but again this requires a fair amount of manual, low-level implementation.

Incorporation of Facts and States Composition-operator-based languages do not offer any support for querying non-event data such as facts in a database or states of objects and other entities. In part this is due to their origins in active databases, where composition-operator-based event queries would be used in the E-part of ECA rules, and database data only accessed in the C-part. This separation however precludes the use of views that enrich events by accessing database tables. Data stream query languages allow to query database relations together with event streams and, since both the data stream queries and database queries rely on SQL, this does not even require a switch between query languages. Some data stream query languages also allow to access objects of some host programming language (e.g., in the case of Esper [Esp], Java objects). Production rules excel at incorporating facts and states into event queries, since they are primarily intended for state-oriented processing of facts rather than processing of events. When tables in a database are to be accessed, this has to be done through the mechanisms offered by the host programming language. In contrast to data stream query languages, production rules can also directly modify states by updating the working memory as part of their actions.

3.5.2 Semantics and Language Design

The second aspect for our comparison looks at issues concerning the languages as a whole such as semantics or ease-of-use.

Formal Semantics Current composition-operator-based languages have a tendency to lack formal semantics, or only provide formal semantics that ignore aspects relating to data in events. Data stream query languages have very precise semantics, but these semantics rely on a somewhat unintuitive round-trip from event streams to relations. The semantics of production rules have a strong imperative flavor, mainly specifying when which rule will execute. In particular when conflict resolution between several rules that can fire comes into play, these semantics can get rather complicated. Further the semantics always rely on the host programming language for actions, and this host programming language usually is an imperative and Turing-complete language.

Ease-of-Use and Learning Curve Composition operators are easy to use, intuitive, and learned very quickly. They also provide a very compact notation for complex event queries. However, some operators are prone to potential misunderstandings and might thus lead to surprises for programmers. Data stream query languages are less intuitive because understanding them involves the round-trip from event streams to relations. Since data stream query languages leverage SQL, familiarity with SQL is necessary for their learning. Accordingly learning curves can be expected to be quite different for programmers already familiar with SQL and programmers not knowing SQL. Production rules as such are not difficult to learn, but their use for event processing requires significant experience. Many aspects of event queries must be programmed manually. In particular, the programmer has to think about manual garbage collection. Note that ease-of-use of any of the three styles of event query languages always depends heavily on the types of queries that one wants to program.

Occurrence and Detection Times of Events While composition-operator-based languages started out using time points as occurrence times of simple and complex events, it has later become more fashionable to use time intervals. The occurrence time of a complex event then is the time interval covering all constituent events. In most cases it is clear by looking at a query when a complex event will be detected. However there are some cases involving, e.g., relative temporal

events, where the detection time is not always obvious. Data stream query languages use only time points for the occurrence times of simple and complex events. The detection time of a complex event is more difficult to determine by looking at a query because it depends on the relation-to-stream operator. The relation-to-stream operator in turn might depend on the difference between the current and the previous state of the result relation. With production rules, all issues related to the representation of time and the detection time of complex events are left to the programmer.

Extensibility and Flexibility Extending a composition-operator-based language would typically consist in defining a new composition-operator. Since this actually means changing the language and involves adapting its implementation, composition-operator-based languages are not easy to extend. Data stream query languages often offer some limited support for user-defined extensions, in particular user-defined windows and user-defined functions. Production rules are very flexible and easily extended due to their close cooperation with a host programming language.

3.5.3 Environment

Our third and final aspect for comparing the three different approaches looks at their use and implications in a larger environment.

Data Model and Integration There is no specific data model common to all composition-operator-based languages. Many languages neglect the data aspect of event queries; those languages that pay attention to event data often assume an event to consist of attribute-value pairs together with an event type. Very few languages offer direct support for events represented as XML messages.[13] Composition-operator-based languages are traditionally integrated into an active database system, but more recently they are also used in ECA rule languages that are separate from a database system. Data stream query languages use the relational data model for events, i.e., each event is a tuple in a specific stream (corresponding to the event type) and has time stamp. Support for events represented as XML messages is fairly limited, typically requiring a conversion into relational data through a mechanism like SQL/XML [EMK+04]. Data stream query languages are often tightly integrated with a database system, so as to access non-event data in the database. However, some languages are also integrated with a regular programming language (e.g., Esper [Esp] integrates with Java and can query events represented as Java objects). Production rule languages use the data model of their host programming language and integrate tightly into this host language.

Availability and Quality of Implementations While composition-operators have been popular in research, not many language implementations are available. We are only aware of two products implementing a composition-operator-based language, ruleCore [MS] and AMiT [AE04]. For data stream query languages, there are a number of fairly recent commercial and open source products. Since many commercial products address high-end markets such as algorithmic trading where speed is essential, these products offer very good scalability and performance. Products for production rules have been around for some time. In terms of scalability and performance, these products are typically limited, especially compared to products for processing data streams. The rete algorithm used in these systems is quite memory-intensive. Further the conflict resolution in production rule languages leads to a computational overhead. Since event processing applications often make little use of conflict resolution, the effort that goes into it can be deemed wasteful.

Development Tools Both composition-operator-based languages and data stream query languages do not offer very extensive development tools at present. There are some user interfaces where queries can be specified by filling out forms or graphically in a data flow network. Mostly

[13] Note that while the title of [BKK04] contains "XML," events there are DOM events [Pix08] that are generated through interaction with an XML document (e.g., in a browser) and not XML messages.

it is still required that a programmer writes queries directly in textual syntax. With the commercialization of CEP, however, it can be expected that more tool support will be available in the future. Commercial production rule systems typically offer very strong development tools. However these are not tailored towards event processing. Particularly interesting is that many production rule systems offer tools to develop domain-specific languages. A domain-specific language (DSL) allows users to specify their rules in a controlled natural language using the concepts and vocabulary of the business domain at hand, ideally without needing much technical background knowledge. However, developing a DSL is an investment requiring significant effort from technical experts. Further, we are not aware of any experiences so far with applying DSLs in event processing contexts.

3.5.4 Summary

The table in Figure 3.7 summarizes the comparison of this section. The entries in the table use the scale $++$, $+$, 0, $-$, $--$ to indicate how well each approach supports a certain feature. As with any such table, some entries will be generalizations; we refer to the previous text for deeper discussion and explanations.

3.6 Hybrid Approaches

To complete our state-of-the-art survey, we now discuss some hybrid approaches that attempt to combine the different styles of querying complex events.

3.6.1 Pattern Matching in Data Stream Query Languages

The original model of CQL provides only stream-to-relation, relation-to-relation, and relation-to-stream operators. Pattern-matching on streams can be added to this model as stream-to-stream operator. Such an operator allows to specify and match patterns that resemble regular expressions (for pattern matching in strings) in data streams. These patterns have some similarities with queries formed using event composition operators, which is why we classify pattern matching in data stream query languages as a hybrid approach here.

Such operators are particularly useful in processing market data such as stock ticks. There it is often important to recognize certain shapes in the graph of price ticks. For example, one might look for situations where a price falls over a span of several price ticks and then rises, falls again, and rises again, typically with some thresholds and relations between the minimum and maximum prices. In terms of the corresponding price graph, one is looking for a "W"-shape with this query. Such a query can be written as a regular expression $F{+}R{+}F{+}R{+}$, when F stands for a falling price event (i.e., the previous price tick was higher then the current) and R for a rising price event (i.e., the previous price was lower then the current).

Such pattern matching is recently supported in Oracle's CEP product [Ora]. The idea and efficient techniques for evaluating such pattern matching operations have been considered earlier in [SZZA01, SZZA04]. Esper also provides some more restricted pattern matching operations on streams [Esp].

3.6.2 Composition Operators on Top of Data Stream Queries

Another approach for combining the capabilities of composition-operators with the capabilities of a data stream query language is described in [GAC06, CA08]. Incoming data streams are first processed with a data stream query language, and then the output events are further processed with a composition-operator-based language to detect the complex events desired by the application. Note that this work does not propose an integrated language that has the capabilities of both composition-operators and data stream queries; rather it proposed an architectural model for using both within a single application and thus combining their respective strength.

	Composition operators	Data stream lang.	Production rules
Temporal aspects	++	0 (temporal windows)	− (via host prog. lang.)
Negation	+	0 (inconvenient in SQL)	− (temp. aspects)
Aggregation	−−	++	− (temp. aspects)
Consumption and selection	++	−−	0 (manually control)
Facts and States	− (C-part of ECA rules)	+	++
Formal Semantics	0 (data not considered)	+ (precise but unintuitive)	− (essentially imperative programming)
Ease-of-Use, Learning Curve	+ (misinterpretations of operators possible)	0 (conversion between streams and relations)	− (e.g., manual garbage collection)
Occurrence and detection time	+	− (depends on relation-to-stream op.)	−− (left to programmer)
Extensibility, flexibility	−	+ (user-defined functions and windows)	++ (host prog. lang.)
Data model: XML support	−− (with exceptions)	+ (via SQL/XML)	− (conversion to objects)
Integration	Active database or stand-alone	Database; sometimes prog. lang.	Prog. lang. (e.g., Java)
Implementations	0 (mainly prototypes)	++ (highly scalable)	+ (scalability issues)
Development tools	−	−	0 (not tailored for CEP)

Figure 3.7: Summary of the comparison between composition operators, data stream langauges, and production rules for querying complex events

In comparison to the previous hybrid approach of adding pattern matching to a data stream query language, this approach can be thought of being the other way round: it first applies data stream queries and then the pattern matching through composition operators, whereas the previously discussed approach first applies pattern matching and then data stream queries.

3.6.3 Event Composition Operators in Production Rules

Work in [WBG08] aims at adding event composition operators to a production rule system based on the rete algorithm. Events occur over time intervals and the supported composition operators are based on Allen's Interval relations. So far, the work offers so far no operators for negation, aggregation, counting, or similar advanced queries. The composition operators are implemented as beta-nodes in the rete network.

Additionally, the composition operators support metric temporal constraints that pose limitations on the distance between the start or end points of the occurrence time intervals. These limitations can be upper bounds or lower bounds on the distance as well as exact specifications. The metric temporal constraints are used to enable an automatic garbage collection of events that become irrelevant (cf. also Chapter 15).

Chapter 4

Background: Xcerpt and XChange

The event query language XChange^{EQ}, which is developed in this thesis and will be presented in the following chapters, caters for specifics of Web data and reactivity on the Web. To this end, it builds upon two existing projects: Xcerpt, a rule-based Web query language, and XChange, a reactive rule language for the Web. This section gives an introduction into these two languages, as necessary for understanding XChange^{EQ}.

4.1 Xcerpt: Querying and Reasoning on the Web

Xcerpt [SB04, Sch04] is a declarative, rule-based query language for Web data as well as other kinds of semi-structured data. It is used in the event query language XChange^{EQ} for querying data in simple events that are transmitted as XML messages.

4.1.1 Distinctive Features

Xcerpt has a number of features that distinguish it from the current standard Web query languages XSLT, XQuery, and SPARQL, which have been introduced shortly in Chapter 2.2.1. Some of these feature also make Xcerpt particularly suitable as a basis for an event query language.

Pattern-Based Approach Queries in Xcerpt are specified as patterns for the data that is accessed to extract interesting portions from. Similarly, data that is to be newly constructed as the result of a query is also specified by patterns. The patterns closely resemble data and can be thought of as forms or templates for the data.

This pattern-based approach where queries are in close correspondence to data gives rise to a language that is fairly intuitive and easy-to-use with a human-friendly syntax. Queries can be written by cut-and-pasting fragments of example data for input and result and successively modifying these fragments into patterns that either extract data (in the case of an example for the input) or construct new data (in the case of an example for the result). The patterns also give rise to a visual language called visXcerpt [BBS03, BBSW03] that realizes the vision of a close correspondence between visual and textual syntax.

For querying simple events that are received as XML messages, the pattern-based approach has a salient advantage, because querying simple events is actually a two-folded task: one has to (1) specify a class of relevant events (e.g., all order events; this corresponds to the event type, cf. Chapter 3) and (2) extract data from the events (e.g., the customer name and item number). As we will see, the patterns of Xcerpt serve both purposes well since they both describe the structure of data and bind variables.

Separation of Extraction and Construction of Data Xcerpt clearly distinguishes and separates patterns that access existing data to extract relevant portions of it (so-called query terms)

and patterns that construct new data (so-called construct terms). In contrast, XQuery and XSLT mix and nest the extraction of data (e.g., `for` or `let` statements in XQuery) and the construction of new data (e.g., `return` statement in XQuery).

For querying events, a separation of the intrinsic query (where data is only extracted without constructing new data; also sometimes called the "query proper") and the construction proves beneficial. It allow an author to first focus on the events that are to be detected over time together with their data and relationships, and then separately on the result that should be generated upon detection.

Rules and Reasoning Query (proper) and construction are brought together in deductive rules; the rule body ("if"-part or antecedent) contains a query, the rule head ("then"-part or consequent) contains a construction. When a rule is applied, we conceptually first evaluate the query in its body. If this is successful, i.e., the patterns specified in the query can be matched to existing data, then new data is constructed according to the specification in the head. Information flows from the rule body to the head in form of variable bindings.

Such rules give rise to deductive reasoning (see also Chapter 2.2.2): a rule can query results constructed by other rules (including itself) and construct new results from it. They also provide an abstraction mechanisms and are convenient for mediating data of different schemas.

Rules can be argued to be as important for events as they are for regular, non-event data. It is therefore conceptually convenient to base XChange^{EQ} on an existing rule language, even though rules about events require some significant changes to the approaches used for rules about regular, non-event data.

Versatility Xcerpt aims at being versatile with respect to data formats and models, allowing to access and construct data in different formats even within a single query [BFB$^+$05]. In particular it aims at making it easy to query both XML and RDF data, as needed for example for querying both XML documents and RDF meta data that is associated with the documents (e.g., through GRDDL [Con07, BFHL07]). In contrast, most existing query languages for Web data support only a single data format and model (e.g., XML for XQuery and XSLT, RDF for SPARQL).

Events are often used to signal changes in some data source (e.g., insertion, deletion) and contain fragments of the changed data. Accordingly, versatility can be argued to be as important for event queries on the Web as it is for regular, non-event Web queries.

4.1.2 Data Terms

XML and other Web data is represented in Xcerpt in a term syntax that is arguably more concise and readable than the original formats, in particular when considering also query terms and construct terms (Sections 4.1.3 and 4.1.4). The term syntax also provides two features that are not found in XML: First, child elements in XML are always ordered. Xcerpt allows children to be specified as either ordered or unordered, the latter bringing no added expressivity to the data format but being interesting for efficient storage based on reordering elements and for avoiding incorrect queries that attempt to make use of an order that should not exist. Second, the data model of XML is that of tree. Xcerpt is more general supporting rooted graphs, which is necessary to transparently resolve links in XML documents (specified, e.g., with IDREFs [B$^+$06a, B$^+$06b] or with XLink [DMO01, BE05b, BE05a]) and to support graph-based data formats such as RDF.

Figure 4.1(a) shows an Xcerpt data term for representing information about flights; its structure and contained information corresponds to the XML document shown in Figure 4.1(b). A data term is essentially a pre-order linearization of the document tree of an XML document. The element name, or *label*, of the root element is written first, then surrounded by square brackets or curly braces, the linearizations of its children as subterms separated by commas. Square brackets `[ ]` indicate that the order of the children is relevant and must be preserved. Curly braces `{ }` indicate that the order of children is irrelevant. In the example of Figure 4.1(a), the order of the `flight` children of the `flights` element is indicated as relevant, whereas the order of the children of the `flight` elements is not.

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

  flight {
    number { "LH3862" },
    from   { "MUC" },
    to     { "FCO" }
  },

  flight {
    number { "LH3863" },
    from   { "FCO" },
    to     { "MUC" }
  }
]
```

(a) Data term

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<flights>
  <flight>
    <number>UA917</number>
    <from>FRA</from>
    <to>IAD</to>
  </flight>

  <flight>
    <number>LH3862</number>
    <from>MUC</from>
    <to>FCO</to>
  </flight>

  <flight>
    <number>LH3863</number>
    <from>FCO</from>
    <to>MUC</to>
  </flight>
</flights>
```

(b) XML document

Figure 4.1: An Xcerpt data term and its corresponding XML document

```
flights {{
  flight {{
    to   { var D },
    from { "MUC" }
  }},
}}
```

(a) Destinations from MUC

```
flights {{
  desc number { var N }
}}
```

(b) All flight numbers

```
flights {{
  var F -> flight {{
    number {{ var N }}
    without to {{ "MUC" }}
  }}
}}
```

(c) Flights *not* going to MUC

Figure 4.2: Examples of Xcerpt query terms

The data term syntax of Xcerpt also accommodates for graph edges beyond the tree-structure, for other entities than element and text nodes (e.g., attributes), namespaces, etc. However for understanding XChange^{EQ} in the scope of this thesis, these features are not necessary and we therefore refer to [SB04, Sch04] for more details.

4.1.3 Query Terms

A query term describes a pattern for data terms; when the pattern matches, it yields (a set of) bindings for the variables in the query term. Variable bindings are also called substitutions, and sets thereof substitution sets. The syntax of query terms resembles the syntax of data terms and extends it to accommodate variables, incompleteness, and further query constructs.

(Unrestricted) variables Variables in query terms are indicated by the keyword `var`. They serve as placeholders for arbitrary content and keep query results in the form of bindings. Figure 4.2(a) shows a query term that extracts all possible direct destinations from Munich (`MUC`) from a data term or document like the one in Figure 4.1. In the example there is only one variable, D, and the result of evaluating the query term is a set of bindings for this variable. For the example input data term of Figure 4.1, the result is $\{\{D \mapsto \texttt{"FCO"}\}\}$, i.e., there is only a single binding. Note that an empty set would signify that the query term and data term do not match.

Complete and incomplete subterm specification In the patterns of query terms, single brackets or braces indicate a complete specification of subterms. In order for such a pattern to match, there must be a one-to-one matching between subterms (or children) of the data term and

the query term. Double brackets or braces in contrast indicate an incomplete specification (w.r.t. to breadth): each subterm in the query term must find a match in the data term, but the data term may contain further subterms. As with data terms, square brackets indicate that the order of subterms is relevant to the query and curly braces that it is not. According to the query term in the example, `flight` data terms must contain a subterm `from` and a subterm `to` and may contain further subterms (e.g., `number`). In contrast, `from` may only have a single child `"MUC"`, no further children, and `to` also may only have a single child (but of arbitrary content).

Incompleteness in depth Incompleteness in depth, that is matching subterms that are not immediate children but descendants at arbitrary depth, is supported with the construct `desc`. The query term in Figure 4.2(b) extracts all flight numbers by searching for `number` elements at arbitrary depths. The result for the example input contains three bindings for variable N:

$$\{\ \{N \mapsto \texttt{"UA917"}\},\ \{N \mapsto \texttt{"LH3862"}\},\ \{N \mapsto \texttt{"LH3863"}\}\ \}$$

Variable restrictions Variables can also be restricted using the “as” construct written using an arrow (`->`) in the form `var X ->` q (with a query term q). The variable then does not match arbitrary content like unrestricted variables, but only content that matches q. Restricted variables are in particular useful for extracting whole subtrees (or subterms) from an XML document. The variable F in Figure 4.2(c) is bound to `flight` subterms that match the specified pattern.

Subterm negation Patterns can also contain negations, that is the negated subterm may not occur in the data. This is specified with the `without` keyword. Figure 4.2(c) locates all flights (variable F) together with their flight numbers (variable N) that do `not` go to Munich (`MUC`), i.e., do not have a subterm `to { "MUC" }`. For the example input, the query gives two bindings for the variables:

$$\begin{aligned}\{\ \ &\{F \mapsto \texttt{flight \{ number \{"UA917"\}, from \{"FRA"\}, to \{"IAD"\} \}},\quad N \mapsto \texttt{"UA917"}\},\\ &\{F \mapsto \texttt{flight \{ number \{"LH3862"\}, from \{"MUC"\}, to \{"FCO"\} \}},\quad N \mapsto \texttt{"LH3862"}\}\quad \}\end{aligned}$$

Further constructs Xcerpt query terms also cater for optional subterms (`optional`), label variables, positional variables (`pos`), regular expression matching, non-structural conditions such arithmetic comparisons (`where`) and more. These construct will not be detailed here but discussed when necessary in examples of XChange^{EQ} event queries.

4.1.4 Construct Terms and Single Rules

Construct terms are used to create new data terms using variable bindings obtained by a query. A construct term describes a pattern for the data terms that are to be constructed. The syntax of construct terms resembles the syntax of data terms and extends it to support variables and grouping.

Rules Construct terms and queries are connected through rules of the form `GOAL` c `FROM` q `END` or `CONSTRUCT` c `FROM` q `END`. In both cases, c is a construct term and q a single query term or formula built from several query terms. `GOAL` rules directly generate output, while `CONSTRUCT` rules are used for intermediate results in rule-based reasoning that are not in the output (see next section).

Variables In constructing new data, variables in construct terms are simply replaced by the bindings obtained from the query. The result is a new data term. If there are no grouping constructs, then a new data term is generated for each binding of the variables. The construct term of the rule in Figure 4.3(a) will construct one query term *for each* non-stop destination from Munich (`MUC`). Note that the query in the `FROM` part of the rule is the same as used previously in Figure 4.2(a).

```
GOAL
  muc-dest [ var D ]
FROM
  flights {{
    flight {{
      to   { var D },
      from { "MUC" }
    }},
  }}
END
```

(a) One term per destination

```
GOAL
  ul [
    all li [ var D ]
    group by { var D }
  ]
FROM
  flights {{
    flight {{
      to   { var D },
      from { "MUC" }
    }},
  }}
END
```

(b) Single list of destinations

```
GOAL
  table [
    all tr [
      td [ var S ],
      td [ count(all var D) ]
    ] order by (lexical) [var S]
  ]
FROM
  flights {{
    flight {{
      to   { var D },
      from { var S }
    }},
  }}
END
```

(c) Numbers of destinations, sorted

Figure 4.3: Examples of Xcerpt construct terms

Grouping Constructing a separate term for each variable binding is fairly limited and more complex restructuring of data is often needed. In particular data must be grouped together in a single term. In our example, we might want a single term that contains all non-stop destinations instead of separate terms, e.g., as a list in XHTML markup. Such grouping can be expressed as a subterm in a construct term of the form `all` c `group by` { `var` V }. Its effect is to generate a subterm from c for each distinct binding of the variable V. The construct term in Figure 4.3(a) will thus generate a single `ul` term that contains multiple `li` subterms, one for each non-stop destination (variable D). The `group by` part can be left out in this example and many other cases. The default then is to group by the free variables immediately inside the construct term after `all`.

Nested Grouping More complicated structures can be built by nesting grouping constructs. The construct term in Figure 4.3(c) produces a table in XHTML markup. Its first column contains airports that have at least one flight leaving this airport. The outer grouping generates a table row (`tr`) for each airport; when an airport is the origin of several flights, it still leads to only one table row. In doing so, the set of all variable bindings is divided into groups where the variable S has the same value. Each such group of variable bindings is then used by the inner grouping for the respective table row (`tr`).

Sorting and Nesting When grouping generates a list, the order of the generated subterms can be influenced with an `order by` clause. The construct term in Figure 4.3(c) sorts the table rows alphabetically by the airport codes (first column).

Aggregation Grouping is also used in conjunction with aggregation functions such as minimum, maximum, sum, or count. The inner grouping in Figure 4.3(c) counts the number of different non-stop destinations from the airport in the first column and puts this value into the second column. The overall result of the rule in Figure 4.3(c) therefore is an alphabetically sorted table of airports (first column) with the respective number of different non-stop destinations (second column).

Further constructs There are some further construct which will no be detailed here. Instead of producing a subterm for all different bindings of a variable, `some` allows to restrict this to a fixed number. When `optional` is used for a variable in the query of a rule, then it must also be used for that variable in the construct term; an accompanying `with default` allows to specify a value that is used in case the variable has no binding. Issues related to constructing graphs rather than trees are discussed mainly in [Fur08].

```
CONSTRUCT
  connection {
    from { var F },
    to   { var T }
  }
FROM
  desc flight {{
    from { var F }
    to   { var T }
  }}
END
```

(a) Base case

```
CONSTRUCT
  connection {
    from { var F },
    to   { var T }
  }
FROM
  and {
    desc flight {{
      from { var X },
      to   { var T }
    }},
    connection {
      from { var F },
      to   { var X }
    }
  }
END
```

(b) Transitive closure

```
GOAL
  table [
    all tr [
      td [ var F ],
      td [
        ul [
          all li [ var T ]
        ]
      ]
    ]
  ]
FROM
  connection {
    from { var F },
    to   { var T }
  }
END
```

(c) Generation of output

Figure 4.4: Rule-based reasoning with Xcerpt: program to find all connections

4.1.5 Reasoning with Rules

The rules in Figure 4.3 were fairly simple, each containing only a single query term and immediately generating output. For more complicated queries, boolean connectives (`and`, `or`, `not`) can be used to build formulas from query terms. Further Xcerpt supports rule-based reasoning, where results of a rule can recursively be used by other rules (including the rule itself). As mentioned before, rules are that used just for inference without generating output start with the keyword `CONSTRUCT`.

The three rules in Figure 4.4 use rule-based reasoning to find all connections between airports using connecting flights. This reasoning task has already been mentioned in Chapter 2.2.2. The first rule (Figure 4.4(a)) states that if there is a flight between two airports, then there is a connection between these two airports. The second rule (Figure 4.4(b)) realizes the inference based on transitive closure: if there is a flight from A to B and a connection from B to C, then there is a connection from A to C. The third rule (Figure 4.4(b)) uses the information generated by the other two rules to generate output. The result is a table in XHTML markup where the first column contains airports. The cell in the second column contains a list of all possible destinations that can be reached with connections of arbitrary length.

Note that such a program might not terminate in a simple backward-chaining evaluation when data is cyclic. However, forward-chaining or more involved backward-chaining with memoization could be used to evaluate this program in a terminating manner.

Like most rule languages, Xcerpt puts some limits on recursion to avoid query programs that might be unintuitive and semantically difficult. In the simplest case [Sch04], programs must be stratifiable with respect to negation and grouping, that is, they must not have recursive cycles that involve negation or grouping. More liberal approaches have also been developed in [BS03] and [Est08]. Stratification will be explained in Chapters 6 and 10 in more detail for XChange$^{\mathrm{EQ}}$.

4.2 XChange: Reactivity on the Web

XChange [BBEP05, Pǎt05, BEP06b, BEP06c] is a reactive rule language for the Web. It addresses challenges such as updating Web data in response to events, propagating updates over the Web, and realizing simple workflows. The event query language XChange$^{\mathrm{EQ}}$ embeds into XChange as a sublanguage for detecting relevant (complex) events.

4.2.1 Building Blocks of Reactivity

Reactivity can be described as the ability to detect events and respond to them automatically in a timely manner. On the Web, reactivity bridges the gap between a passive Web, where

data sources can only be accessed to obtain information, and a more dynamic, "reactive" Web, where data sources change (evolve) in reaction to events bringing in new information or rendering existing information out-of-date. Reactivity spans a broad field from Web applications such as e-commerce platforms that react to user input (e.g., putting an item into the shopping basket), over Web services that react to notifications or service requests (e.g., SOAP messages), to distributed Web information systems that react to updates in other systems elsewhere on the Web (e.g., update propagation among scientific Web databases).

XChange keeps with the basic assumptions of the Web. Nodes are identified by URIs. Each node is autonomous and there are no centralized authorities, adhering to the Web decentralized nature. However, XChange makes the assumption that nodes on the Web inform other interested nodes about relevant events by sending appropriate messages. Events are communicated directly and asynchronously in a push-manner as XML messages over HTTP. Push communication has several advantages over pull communication: it allows faster reaction, avoids unnecessary network traffic through periodic polling, and saves local resources.

An XChange program runs locally at a single Web node and reacts to incoming events. By sending and receiving events it can coordinate its behavior with that of other Web nodes. These Web nodes might realize their reactive behavior also as XChange programs or in any other way that allows for message-based communication.

Typical reactions to events are updates to persistent data local to the Web node, requests for updates at remote Web nodes, or sending event messages to other Web nodes. Both persistent data and event messages are represented in XML (or by extension other Web data formats). XChange is well-suited for realizing local updates, propagation of updates in a distributed Web information system, and simple workflows. These applications will be discussed in Section 4.2.6.

4.2.2 Event-Condition-Action (ECA) Rules

An XChange program consists of one or more reactive rules of the form `ON` *event query* `IF` *Web query* `DO` *action*.[1] Such an ECA rule has the following meaning: When events matching the event query are received and the Web query is successfully evaluated, then the action is performed. Both event query and Web query can extract data through variable bindings, which can then be used in the action. As we can see, both event and Web queries serve a double purpose of detecting *when* to react and influencing —through binding variables— *how* to react. For querying data, as well as for updating data, XChange embeds and extends the Web query language Xcerpt presented earlier.

Figure 4.5 shows an example of an XChange ECA rule, which will be used for our subsequent explanations. The individual parts of the rules employ Xcerpt and its pattern-based approach. Patterns are used for querying data in both the event and condition part, for constructing new event messages in the action part, and for specifying updates to Web data in the action part.

4.2.3 Events

Event messages Events in XChange are represented and communicated as XML messages. The root element is for all events `xchange:event`, where the prefix `xchange` is bound to the XChange namespace. Events messages also carry some meta-data as children of the root element such as `raising-time` (i.e. the time of the event manager of the Web node raising the event), `reception-time` (i.e. the time at which a node receives the event), `sender` (i.e. the URI of the Web node where the event has been raised), `recipient` (i.e. the URI of the Web node where the event has been received), and `id` (i.e. a unique identifier given at the recipient Web node). An example event that might represent the cancellation of a flight with number "UA917" for a passenger named "John Q Public" is shown in both XML and term syntax in Figure 4.6.

[1] In the course of the development of XChange, different keywords and orders for the rules have also been used. In particular, rules can also be written as `RAISE` *event raising action* `ON` *event query* `FROM` *Web query* or `TRANSACTION` *update action* `ON` *event query* `FROM` *Web query*.

```
ON
   xchange:event {{
     flight-cancellation {{
       flight-number { var N },
         passenger {{
           name { "John Q Public" }
   }} }} }}
IF
   in { resource { "http://www.example.com/flights.xml", "xml" },
     flights {{
       flight {{
         number { var N },
           from   { var F },
           to     { var T }
   }} }} }
DO
  and {
    xchange:event [
      xchange:recipient [ "http://sms-gateway.org/us/206-240-1087/" ],

      text-message [
        "Hi, John! Your flight ", var N,
        " from ", var F, " to ", var T, " has been canceled."
    ] ],

    in { resource { "http://shuttle.com/reservation.xml", "xml" },
      reservations {{
        delete shuttle-to-airport {{
          passenger { "John Q Public" },
            airport   { var F },
            flight    { var N }
    }} }} }
END
```

Figure 4.5: An XChange ECA rule reacting to flight cancellations for passenger “John Q Public”

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

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

(a) XML syntax

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

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

(b) Data term syntax

Figure 4.6: Example of an event message

Simple ("atomic") event queries The event part of a rule specifies a class of events that the rule reacts upon. This class of events is expressed as an event query. A simple (or atomic) event query is expressed as a single Xcerpt query term.

Event messages usually contain valuable information that will be needed in the condition and action part of a rule. By binding variables in the query term, information can flow from the event part to the other parts of a rule. Hence, event queries can be said to satisfy a dual purpose: (1) they specify classes of events the rule reacts upon and (2) they extract data from events for use in the condition and action part in the form of variable bindings.

An XChange program continually monitors the incoming event messages to check if they match the event part of one of its XChange rules. Each time an event that successfully matches the event query of a rule is received, the condition part of that rule is evaluated and, depending on the result of that, the action might be executed.

The event part of the ECA rule from Figure 4.5 would be match the event message in Figure 4.6. In the condition and action part the variable N would then be bound to the flight number "`UA917`".

Complex ("composite") event queries To detect complex events, the original proposal of XChange supported composition operators (cf. Chapter 3.2) such as `and` (unordered conjunction of events), `andthen` (ordered sequence of events), `without` (absence of events in a specified time window), etc. [Eck05, Pătt05, BEP06a, BEP06b]. These composition operators share the weaknesses and associated problems of other composition-operator-based event query languages discussed in Chapter 3.2. The work of this thesis, XChange$^{\text{EQ}}$, seeks to replace these composition operators with an improved and radically different approach to querying complex events.

4.2.4 Conditions

Web queries The condition part of XChange rules queries data from regular Web resources such as XML documents or RDF documents. It is a regular Xcerpt query, i.e., anything could come after the `FROM` part of an Xcerpt rule. Like event queries in the event part, Web queries in the condition part have a two-fold purpose: they (1) specify conditions that determine whether the rule's action is executed or not and (2) extract data from Web resources for use in the action part in the form of variable bindings.

The condition part in the rule from Figure 4.5 accesses a database of flights like the one from Figure 4.1 located at `http://www.example.com/flights.xml` (the resource is specified with a URI using the keyword `in`). It checks that the number (variable N) of the canceled flight exists in the database and extracts the flight's departure and destination airport (variables F and T, respectively).

Deductive rules Web queries can facilitate Xcerpt rule chaining (as introduced in Section 4.1.5). For this, an XChange program can contain Xcerpt `CONSTRUCT-FROM` rules in addition to its ECA rules. Such rules are useful for example to mediate data from different Web resources. In our example we might want to access several flight databases instead of a single one and these might have different schemas. Deductive rules can then be used to transform the information from several databases into a common schema.

4.2.5 Actions

The action part of XChange rules has the following primitive actions: rasing new events (i.e., creating a new XML event message and sending it to one or more recipients) and executing simple updates to persistent data (such as deletion or insertion of XML elements). To specify more complex actions, compound actions can be constructed from these primitives.

Raising new events Events to be raised are specified as a construct terms for the new event messages. The root element of the construct term must be labeled `xchange:event` and contain

at least on child element `xchange:recipient` which specifies the recipient Web node's URI. Note that the recipient can be a variable bound in the event or condition part.

The action of the ECA rule in Figure 4.5 raises (together with performing another action) an event that is sent to an SMS gateway. The event will inform the passenger that his flight has been canceled. Note that the message contains variables bound in the event part (N) and condition part (F, T).

Updates Updates to Web data are specified as so-called update terms. An update term is a (possibly incomplete) query pattern for the data to be updated, augmented with the desired update operations. There are three different types of update operations and they are all specified like subterms in an update term. An insertion operation `insert` c specifies a construct term c that is to be inserted. A deletion operation `delete` c specifies a query term q for deleting all data terms matching it. A replace operation `replace` q `by` c specifies a query term q to determine data items to be modified and a construct term c giving their new value. Note that update operations cannot be nested.

Together with raising a new event, the action of the ECA rule in Figure 4.5 modifies a Web resource containing shuttle reservations. It removes the reservation of our passenger's shuttle to the airport. The update specification employs variables bound in the event part (N) and condition part (F).

Due to the incompleteness in query patterns, the semantics of complicated update patterns (e.g., involving insertion and deletion in close proximity) might not always be easy to grasp. Issues related to precise formal semantics for updates that are reasonably intuitive even for complicated update terms have been explored in [Coş07]. So-called snapshot semantics are employed to reduce the semantics of an update term to the semantics of a query term.

Compound Actions Actions can be combined with disjunctions and conjunctions. Disjunctions specify alternatives, only one of the specified actions is to be performed successfully. (Note that actions such as updates can be unsuccessful, i.e., fail). Conjunctions in turn specify that all actions need to be performed. The combinations are indicated by the keywords `or` and `and`, followed by a list of the actions enclosed in braces or brackets.

The actions of the rule in Figure 4.5 are connected by `and` so that both actions, the sending of an SMS and the deletion of the shuttle reservation, are executed.

4.2.6 Applications

Due to its built-in support for updating Web data, an important application of XChange rules is local evolution, that is updating local Web data in reaction to events such as user input through an HTML form. Often, such changes must be mirrored in data on other Web nodes: updates need to be propagated to realize a global evolution. Reactive rules are well suited for realizing such a propagation of updates in distributed information portals.

A demonstration that shows how XChange can be applied to programming reactive Web sites where data evolves locally and, through mutual dependencies, globally has been developed in [Gra06] and presented in [BEGP06b, BEGP06a]. The demonstration considers a setting of several distributed Web sites of a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS). ECSS is subdivided into participating universities, thematic working groups, and project management. Universities, working groups, and project management have each their own Web site, which is maintained and administered locally. The different Web sites are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites. For example, Web sites maintain information about personal data of members; a change of member data at a university entails further changes at the Web sites of the management and some working groups.

The propagation of updates and other functionality (e.g., sending of newsletters) of these distributed Web sites are realized as XChange ECA rules. While a similar behavior as the one

in the demo could be obtained with conventional programming languages, XChange provides an elegant and easy solution that also abstracts away issues such as low-level network communication protocols. Evolution of data and reactivity on the Web are easily arranged for by using readable and intuitive ECA rules. Moreover, by employing and extending Xcerpt as a query language, XChange integrates reactivity to events, querying of Web resources, and updating those resources in a single, easy-to-learn language.

XChange ECA rules have also been investigated as way to realize workflows, e.g., in business processes. More details on this can be found in [Rom06, BEPR06].

4.3 Summary

This chapter has introduced the Web query language Xcerpt and the reactive Web language XChange. These two languages are the context in which XChange^{EQ} is being developed: XChange^{EQ}, the topic of this thesis, employs Xcerpt to query XML data in simple events and it can be used inside XChange as a sublanguage for specifying complex events. All three languages, Xcerpt, XChange, and XChange^{EQ}, employ a pattern-based approach for dealing with Web data and together give a suite of languages for realizing common tasks on the Web involving querying and reasoning with regular Web data, reacting to and communicating events, updating Web data, and detecting and reasoning with complex events.

Part II

XChange$^{\mathbf{EQ}}$: An Expressive High-Level Event Query Language

Chapter 5

Language Design

The language design of XChange^{EQ} follows a clear rationale based on a few core principles. At the very heart of XChange^{EQ} is the idea that event queries can be described according to the four dimensions data extraction, event composition, temporal and other relationships between events, and event accumulation (Section 5.1), and that an event query language must separate these four dimensions in order to achieve full expressivity (Section 5.2). With the emergence of the Web as a universal information system, XChange^{EQ} caters for the specific needs of querying and reasoning with events on the Web (Sections 5.3 and 5.4). XChange^{EQ} aims at being a declarative, easy-to-use language (Section 5.5) that comes with clear semantics (Section 5.6). Finally, acknowledging that a single language can often not solve all problems, XChange^{EQ} is designed with extensibility in mind (Section 5.7).

The language XChange^{EQ} has first been presented in [BE06a]. Issues related to its language design (and language design of complex event query languages in general) is also discussed in [BE06b, BE07d, BE07a, BE08b]

5.1 Four Dimensions of Querying Events

Characteristic for applications involving event queries is the need to (1) utilize data contained in the events, (2) detect patterns composed of multiple events (i.e., complex events), (3) reason about temporal and other relationships between events, and (4) accumulate events for negation and aggregation. We can understand these requirements as four complementary dimensions that we call data extraction, event composition, temporal (and other) relationships, and event accumulation. These four dimensions, which will be detailed shortly, must (at least) be considered for querying complex events. How well an event query language covers each of the dimensions gives a practical measure for its expressiveness.

Data extraction Events contain data that is relevant for applications to decide whether and how to react to them. The data of events must be extracted and provided (typically as bindings for variables) to test conditions (e.g., arithmetic expressions) inside the query, combine event data with persistent, non-event data (e.g., from a database), construct new events (e.g., by deductive rules, see Section 5.4), or trigger reactions (e.g., database updates).

Often, events are transmitted as messages in XML formats (cf. Chapter 2.4); examples for such message formats include SOAP [G^{+}03], Common Base Event (CBE) [IBM04], and the Facility Control Markup Language (FCML) [BLO^{+}08]. Data in such XML messages can be semi-structured, i.e., have a quite complex and varying structure. This gives a strong motivation to build upon and embed an already existing XML query language into an event query language. Accordingly, XChange^{EQ} builds upon the XML query language Xcerpt; the advantages for using Xcerpt over the standard XML query languages XQuery and XSLT have been outlined in Chapter 4.1.1.

Event composition To support complex events, i.e., events that consist of several events, event queries must support composition constructs such as the conjunction and disjunction of events (more precisely, of event queries). Composition must be sensitive to event data, which is often used to correlate and filter events (e.g., consider only stock transactions from the *same* customer for composition). Event composition also gives rise to relative temporal events, that is, timer events that are defined relative to another event such as "2 hours after event X." Since reactions to events are usually sensitive to timing and order, an important question for complex events is *when* they are detected. In a well-designed language, it should be possible to recognize when reactions to a given event query are triggered without difficulty.

Temporal (and other) relationships Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour, and A happens before B." Temporal relationships between events can be qualitative or quantitative. Qualitative relationships concern only the temporal order of events (e.g., "shipping after order"). Quantitative (or metric) relationships concern the actual time elapsed between events (e.g., "shipping and order more than 24 hours apart").

Time takes dominating role in event processing, since it affects the timing and order of reactions to complex events. Therefore, we concentrate on temporal relationships in XChange^{EQ}. However, there might also be other relationships between events that are of interest in event queries. With these relationships, there is always a consideration whether they should just be considered as relationships between event data or deserve a special treatment. XChange^{EQ}'s language design is kept extensible so that special treatment of other relationships can be easily added in the same manner as temporal relationships.

For some event processing applications, it is interesting to look at causal relationships, e.g., to express queries such as "events A and B happen, and A has caused B." While temporality and causality can be treated similarly in query syntax, causality raises interesting questions about how causal relationships can be *defined* and *maintained*. This issue will be discussed in Chapter 19.

In event processing applications involving geographically distributed event sources, spatial relationships between events can also be of interest. For example, a query might specify that two events occur within 100 meters of each other. Spatial information about events might, as noted above, be simply considered as ordinary data in events — more so than causality because there are less issues regarding the definition and maintenance of it. However, special treatment of spatial relationships might be of interest, e.g., in systems that involve mobile event sources and require spatio-temporal reasoning [Sch08]. Further, spatial relationships might play a role in distributed query evaluation. These issue will also be discussed in Chapter 19.

Event accumulation Event queries must be able to accumulate events to support non-monotonic query features such as negation of events (understood as their absence) or aggregation of data from multiple events over time. The reason for this is that the event stream is —in contrast to extensional data in a database— unbounded (or "infinite"); one therefore has to define a scope, e.g., a time interval, over which events are accumulated when aggregating data or querying the absence of events. Application examples where event accumulation is required are manifold. A business activity monitoring application might watch out for situations where "a customer's order has *not* been fulfilled within 2 days" (negation). A stock market application might require notification if "the *average* of the reported stock prices over the last hour raises by 5%" (aggregation).

5.2 Separation of Concerns: Expressivity and Ease-of-Use

XChange^{EQ} aims at a high expressive power with a full coverage of all four query dimensions. The four dimensions can be considered orthogonal and complementary. We argue that an expressive event query language should use a language design that treats the querying dimensions separated

from each other in syntax and semantics in order to reach high expressive power as well as ease-of-use.

XChange^{EQ}'s language design realizes a separation of concerns in syntax and semantics with respect to the four query dimensions. Because each individual dimension is well-covered, it can be claimed that XChange^{EQ} reaches a certain degree of expressive completeness. The separation of concerns also yields a clear language design, makes queries easy to read and understand, and gives programmers the benefit of a separation of concerns. It further contributes to XChange^{EQ}'s extensibility.

Experience with other event query languages shows that without such a separation not all dimensions are fully covered. Composition operators mix the event querying dimensions, e.g., in the case of the sequence operator, event composition and temporal relationships are mixed. This leads difficulties in correctly expressing and understanding some event queries and also to a certain lack in expressiveness. Some examples of such difficulties have already been shown in Chapter 3.

In Chapter 17, we will analyze several concrete examples of event queries and compare XChange^{EQ} and its separation of concerns with other event query languages. This comparison will further substantiate the claim that a separation of the four query dimensions is beneficial for ease-of-use and necessary for a high expressivity of an event query language.

5.3 Seamless Integration into the (Reactive) Web

XChange^{EQ} caters for the specific requirements of complex event processing on the Web and is tailored towards a seamless integration into the reactive Web. Because integration and monitoring is a common requirement on the Web, complex event processing can be expected to gain particular momentum in Web-based settings. As discussed in Chapter 2, the Web is becoming the standard infrastructure for information systems, also for those that are intended for a more confined scope than the World Wide Web (e.g., intranets of enterprises). Service-oriented architectures in the enterprise information systems world are commonly based on Web and Web Service standards, but also other application domains such as Supervisory Control and Data Acquisition are employing these standards.

It has become standard on the Web to represent and transmit events as XML messages. Convenient and uncomplicated access to and querying of XML event messages is therefore the first and most important step for making an event query language tailored towards the Web. It should not be necessary to first convert incoming event messages into a different data model (e.g., an object-oriented or relational model) as would be required by most current event query languages. Also, such a conversion might often imply a necessity of strict types and thus go against the idea of supporting semi-structured Web data. XChange^{EQ} natively supports querying XML event messages. To this end and in order to not reinvent the wheel, it embeds the existing Web query language Xcerpt.

Complex event queries are usually not an end in themselves. Rather the primary goal behind detecting complex events is to perform some appropriate reaction. Such reactive behavior can often be expressed in a convenient and declarative manner using a reactive rule language. Accordingly, there is a close connection between complex event queries and reactive rules. XChange^{EQ} pays tribute to this close connection by embedding into the reactive rule language XChange as a sublanguage. The use of event queries as a sublanguage for the E-part of Event-Condition-Action (ECA) rules, entails a need for event queries to be able to return variable bindings that can be used subsequently in the C- and A-part of rules. However, XChange^{EQ} is designed in a way that also allows its deployment as a stand-alone event mediation component in an event-driven architecture [Etz05] or use in Semantic Web ECA frameworks [MAA05a, MAA05b, AA07].

With its close interaction with the Web query language Xcerpt and the reactive Web language XChange, XChange^{EQ} realizes the vision of a seamless integration of event queries, Web queries, and reactive behavior on the Web. The result is a set of cooperating languages that provide, due to the pattern-based approach that is common to all of them, a homogenous look-and-feel.

5.4 Reasoning with Events

XChange^{EQ} supports reasoning with events and complex events based on deductive rules. Deductive rules allow to define new, "virtual" events from the existing ones (i.e., those that are received in the incoming event stream). Deductive rules for events are thus used much in the same fashion as one uses views (or rules) in databases to define new, derived data from existing base data.

Rule-based reasoning about events is highly desirable for a number of reasons: Rules give answer-closedness, i.e., the result is of the same form as the input, allowing for complex events generated by a deductive rule to be further processed by other rules. As such, rules serve as an abstraction mechanism, making query programs more readable, and a mediation mechanism between different schemas for event data. Rules allow to define higher-level application events from lower-level (e.g., database or network) events. This is also useful when reasoning about causal relationships between events, in particular so-called vertical causality [Luc02]. Different rules can provide different perspectives (e.g., of end-user, system administrator, corporate management) on the same event-driven system. Rules give a representation to the complex event that is detected with them and this representation is helpful for testing larger complex event query programs: query logic can now conveniently be considered in isolation of the concrete reactions associated with complex event. Note that despite these good reasons, only very few current event languages support such purely deductive rules (cf. Chapter 3).

Event-based systems usually provide reactive rules, typically Event-Condition-Action (ECA) rules or production rules, to specify reactions to the occurrences of certain events [BBB+07]. While deductive rules can be, and in existing systems often are, implemented using (or "abusing") reactive rules, we argue that deductive event rules are inherently different from reactive rules. They aim at expressing "virtual events," not actions. Accordingly and importantly, deductive rules are free of side-effects. Implementing deductive rules using reactive rules blurs this distinction. This has negative consequences for development and maintainability of the query logic, and restricts optimization: techniques that are applicable for deductive rules such as backward chaining or program rewriting are not generally applicable to reactive rules. Because of the imperative semantics of reactive rules, checks such as detecting cycles that might lead to non-terminating programs or non-stratified uses of negation that are possible for deductive rules are much harder or even impossible for reactive rules.

5.5 Declarative Language and Simplicity

In its language design, XChange^{EQ} seeks to be a declarative query language that values simplicity. Part of its declarative nature that the same expression (in syntax) should have the same meaning (semantics) independently of the context it is used in. This entails that XChange^{EQ} avoids language construct that would place a processing mode or other kind of context from the outside onto expressions and affect their semantics. It further entails that the (declarative) semantics of XChange^{EQ} are stateless.

For regular (non-event) query languages, such a declarative design is common and well-accepted. However there are features in a number of current event query languages that conflict with a declarative nature. For example event instance selection and event instance consumption, as found in some composition-operator-based languages, constitute modes that are applied from the outside to expressions and require stateful semantics for event queries. At a possible price of expressive power, XChange^{EQ} avoids such "undeclarative" language constructs. In the particular case of event instance selection and event instance consumption we also believe that their value is not sufficiently proven from a practical standpoint and other approaches have not been given full consideration so far. This issue is further discussed in Chapter 17.

XChange^{EQ} favors simplicity by trying to keep the number of necessary language constructs at a minimum while still providing a language that is convenient for the human user. This simplicity is aided strongly by the separation of concerns. Instead of supporting a multitude of composition operators for different temporal relationship between the composed events, for

example, XChange^{EQ} uses only a composition of events through a conjunction and then a separated specification of the temporal relationships that must hold between events.

A particular assumption to simplify XChange^{EQ} and its presentation in this thesis is that occurrence times of events are all given according to a single, common time axis. As we have seen in Chapter 2.4.4, this should not be assumed as the general case. However, the language design of XChange^{EQ} has been engineered in a way that extending it to accommodate multiple time axes is fairly straightforward. Hence the simplifying assumption about a single time axis is more of an issue relating to the presentation of the language rather than inherent to its design.

5.6 Semantics

Both declarative and operational semantics are desirable for an event query language, and in this work we provide both for XChange^{EQ}. We particularly aims at leveraging approaches and results that are well-known and explored from traditional query languages and logic programming. By putting event queries on the wheels of traditional queries, it becomes evident where new concepts and methods are needed for querying events — and where they are not needed.

Declarative semantics of XChange^{EQ} aim at being intuitive and natural. As such they work directly with a stream of incoming events and avoid, for example, a round-trip to relations as it is found in the semantics of CQL (cf. Chapter 3.3.5). They further do not require the concept of state. (Note that this cannot be said for the semantics of CQL since some relation-to-stream operators rely on the difference between the previous and current state of a relation.) Declarative semantics of XChange^{EQ} are provided as a (Tarski-style) model theory with accompanying fixpoint theory. This well-known approach from deductive databases [AHV95] and logic programming [Llo93] accounts well for (1) data in events and (2) deductive rules, two aspects sometimes neglected in other event query languages. Declarative semantics of XChange^{EQ} put particular focus on the occurrence time of complex events in recognition of the importance of timing and ordering of reactions to complex events.

Operational semantics of XChange^{EQ} are targeted towards the efficient evaluation of event queries. Note that operational semantics of event queries must, in contrast to the declarative semantics, inherently refer to state since event query evaluation involves storing and garbage collecting events over time. The operational semantics of XChange^{EQ} are grounded in a restricted and extended variant of relational algebra. With this, many topics from database query evaluation such as query rewriting, index structures, join algorithms, and adaptive query evaluation might be reconsidered in the light of event query evaluation. Incremental evaluation of event queries, stateful processing, and garbage collection are accommodated through so-called materialization points.

5.7 Extensibility

XChange^{EQ} puts emphasis on a design that makes the language easily extensible. Given that querying of complex events is still young field where the requirements on languages are not as established as in other areas, we believe this of high importance. Throughout this work we will therefore point out specific opportunities for possible extensions. That XChange^{EQ} can be easily extended is largely due to its separation of dimensions, which allows to extend it in each dimension independently.

Aspects where XChange^{EQ} can be easily extended include the following. New forms of temporal relationships as well as expressions specifying new kinds of temporal events can be easily added. This might be relevant in applications that use domain- or culture-dependent calendars (e.g., calendars including bank holidays for some business applications). In particular, an integration of a calendrical reasoning system such as CaTTS [BRS05] might prove useful for XChange^{EQ}. Similarly, other forms of relationships such as causal or spatial relationships are easy to add. While XChange^{EQ} uses a single time axis for the occurrence times of events, its design keeps the relevance

of multiple time axes in mind and can accommodate them with only little effort. For querying simple events, XChange^{EQ} relies on the Web query language Xcerpt. Extensions to Xcerpt, e.g., for querying other data formats than XML, become immediately applicable to XChange^{EQ}. Further, replacing Xcerpt with another language for querying simple events is possible without turning XChange^{EQ} inside out.

Chapter 6

Syntax and Informal Semantics of XChange$^{\mathrm{EQ}}$

We now introduce the complex event query language XChange$^{\mathrm{EQ}}$. In Chapter 3, we have discussed the three currently prevalent styles of querying events: composition operators, data stream languages, and production rules. XChange$^{\mathrm{EQ}}$ introduces a fourth style where event queries are specified in a way that is somewhat reminiscent of logical formulas. However, XChange$^{\mathrm{EQ}}$ has a human-friendly syntax tailored towards querying events. This chapter is intended as a step-wise, tutorial-like introduction to XChange$^{\mathrm{EQ}}$. It describes the semantics of event queries only informally; formal semantics are developed in the later parts of this thesis. A context-free grammar for XChange$^{\mathrm{EQ}}$ is given in Appendix A. Throughout this section, we use the example of a stock market application.

6.1 Representation of Events

Events and Event Messages Events in XChange$^{\mathrm{EQ}}$ are represented as XML documents or Xcerpt data terms and received by the event processor as messages, which we call accordingly **event messages**. Since we assume a representation for all events that are of interest, we often also use just the term **event** to mean the corresponding event message. Since there are a number of envelope formats for messages in use on the Web (e.g., SOAP, CBE, FCML, cf. Chapter 2.4.2), XChange$^{\mathrm{EQ}}$ does not commit itself to a specific message format but supports arbitrary XML formats for event messages. In this chapter, we will not use any message envelope in order to keep the examples compact and easy to read. Also, we will primarily use the data term representation for events due to its resemblance with query and construct terms.

Where events are not natively represented as XML, as is for example the case with many sensors that use proprietary data formats, events can still be conveniently converted into an XML representation and queried as XML. Note that with appropriate mechanisms in the event processing engine, it might not be necessary for the XML representation of a such an event to be actually materialized (i.e., be constructed as a DOM tree [H^{+}08] in memory or similar). Instead, it might be possible to transform queries into expressions that work directly on the non-XML format.

Occurrence Time Each event has an **occurrence time** associated with it. For simplicity and ease of presentation, we assume that all occurrence times are given according to the same clock or time axis. One easy way to achieve this would be to time stamp events only upon reception of the event message by the event processing engine. Extensions of XChange$^{\mathrm{EQ}}$ to multiple time axes are possible and discussed in Chapter 18. The occurrence time t of an event is a (closed and convex) time interval, i.e., $t = [b, e]$ with time points b and e. Time point b is called its starting time, e

its ending time. In examples we will often just use integers to represent time points. However XChange^{EQ} aims at modeling our natural understanding of real time closely and thus does not assume integers or another discrete domain for time. In particular, there is no assumption of a unique successor for a given time point as in data stream languages like CQL (cf. Chapter 3.3.5). More on the precise requirements for the time domain will be given in Chapter 9. Simple events that are received in the event stream (see below) often happen only at a single time point p not over a time interval. In these cases a degenerate interval $t = [p, p]$ containing only this single time point is used as occurrence time.

We write events as e^t where e is the event message (i.e., a data term) and t the occurrence time (i.e., a time interval $[b, e]$). For example, `hello {"world!"}`$^{[3,7]}$ would be used to express that an event `hello {"world!"}` happens with occurrence time $[3, 7]$.

Event Stream Conceptually, all events arrive in a single stream at the event processing engine. This stream is called the *stream of incoming events* or just the event stream. While there might be several different streams from different event sources, conceptually we can equally use a single stream of incoming events that corresponds to their union. A distinction of multiple streams would only be relevant if either the stream is related to the type of an event or each stream has a separate time axis and the time axes of different streams are not comparable. Using types for streams is not necessary in XChange^{EQ} because this information can equally be modeled as content of the XML event message. For time, we assume only a single time axis.

Events that are received in the incoming event stream are called *base events* to contrast them with *derived events*, that is, events that are generated ("derived") by deductive rules (cf. Section 6.4).

Note that the occurrence time of a base event can be a true time intervals (i.e., one that is not just a time point), although this is not that common in practice.[1] Different events in the event stream may happen at the same time or with overlapping occurrence times.

Example Events In our stock market application, there will be four different types of events: *buy order* events signal that an order to buy some stock has been placed; *sell order* events signal that an order to sell some stock has been placed; *buy* events signal a trade where some stock has been bought; and *sell* events signal a trade where some stock has been sold.

All events are represented as data terms (or equivalently XML documents). Examples of the *buy order* and *buy* events are shown in Figure 6.1. The *sell order* and *sell* events have the same structure, only `buy` labels are replaced with `sell` labels.

The events contain relevant data such as the `customer` involved in an order or transaction and the `stock` that being ordered or changing hands, together with the `volume` (number of stocks). Order events contain a unique identifier `orderId` and the `limit` the customer sets on the price for selling or buying the stock. Trade (buy and sell) events contain also a unique identifier `tradeId` as well as the `orderId` of the order that has lead to the trade. Further they contain the `price` for which the stock has changed hands.

Event sources in our example might generate further events. These would then not affect our examples, unless they have a structure that is close enough to our four events so that they would match any of the simple event queries in our examples.

6.2 Querying Simple Events

Querying simple events, that is events that are represented by a single event message, is a two-fold task: one has to (1) specify a class of relevant events (e.g., all buy events) and (2) extract data from the events (e.g., the price). XChange^{EQ} uses Xcerpt query terms for both specifying classes of relevant events and extracting data from the events. Simple event queries are written in the

[1]It turns out that there is not much to be gained from restricting base events to time points. Further such a restriction could be deemed irritating since it would require different treatment of base events and derived events.

```
<order>
  <orderId>4711</orderId>
  <customer>John</customer>
  <buy>
    <stock>IBM</stock>
    <limit>3.14</limit>
    <volume>4000</volume>
  </buy>
</order>
```

(a) Buy order event in XML syntax

```
order [
  orderId { 4711 },
  customer { "John" },
  buy [
    stock { "IBM" },
    limit { 3.14 },
    volume { 4000 }
  ]
]
```

(b) Buy order event in term syntax

```
<buy>
  <orderId>4711</orderId>
  <tradeId>4242</tradeId>
  <customer>John</customer>
  <stock>IBM</stock>
  <price>2.71</price>
  <volume>4000</volume>
</buy>
```

(c) Buy event in XML syntax

```
buy [
  orderId { 4711 },
  tradeId { 4242 },
  customer { "John" },
  stock { "IBM" },
  price { 2.71 },
  volume { 4000 }
]
```

(d) Buy event in XML syntax

Figure 6.1: Example event messages for the stock market application

form `event` i: q, where `event` is a keyword, q a query term, and i an alpha-numeric identifier. For now, this event identifier i is not interesting, but it will become important later for complex event queries that involve multiple simple event queries.

As explained in Chapter 4.1.3, Xcerpt query terms describe a (possibly incomplete) pattern that is matched against the data. Query terms can contain variables, which will be bound to matching data. The result of matching them against data is a set Σ of variable bindings $\sigma_1, \sigma_2, \ldots$. We also call $\sigma_1, \sigma_2, \ldots$ substitutions and Σ a substitution set. Like Xcerpt, XChange$^{\text{EQ}}$ allows to add a `where` clause to queries in order to specify non-structural conditions, e.g., arithmetic comparisons on values bound in variables.

Example In our stock market application, we might want to recognize "big buy" events, that is buy events with a price total of \$10 000 or more. The price total there is simply the product of the price per stock and the volume. The following simple event query matches such big buy events and extracts some relevant data from the event message into variables:

```
event b: buy {{
  tradeId { var I },
  customer { var C },
  stock { var S },
  price { var P },
  volume { var V }
}}
where { var P * var V >= 10000 }
```

The query term of this simple event queries matches the structure of the buy event in Figure 6.1(d) (but not, for example, of the buy order in Figure 6.1(b)). The arithmetic condition in the `where` clause would also be satisfied for the example buy event in Figure 6.1(d). The result is a substitution set $\Sigma = \{\sigma_1\}$ containing only one substitution $\sigma = \{I \mapsto \texttt{4242}, C \mapsto \texttt{John}, S \mapsto \texttt{IBM}, P \mapsto \texttt{2.71}, V \mapsto \texttt{4000}\}$.

6.3 Absolute Timer Events

CEP applications often refer to specific time points or intervals on the time axis in order to perform certain actions or as part or complex event queries. Our stock market application might want to

perform a specific action every day at 4pm such as sending a message to all customers to remind them that the market will close in an hour. It also might have complex event queries that will determine the average price of a stock over a trading day. Time points such as "every day at 4pm" and time intervals such as "trading day" are usually not represented as messages in the incoming event stream. Still, one needs the ability to query such timer events.

XChange^{EQ} supports queries for timer events in a manner that is syntactically similar to queries against event messages. One can distinguish absolute and relative timer events. Absolute timer events are time points or intervals defined without reference to the occurrence time of some other event. They can be periodic, i.e., a sequence of time points or intervals as in "every day at 4pm." Relative timer events in contrast are defined dependent on some other event. Relative timer events will be discussed in Section 6.7.

Example Queries to absolute timer events are specified just like queries to event messages. In fact, one can imagine querying timer events as querying a "virtual" stream of event messages that represent all possible time points and intervals. XChange^{EQ} offers one built-in way for specifying absolute timer events. The "virtual message" for this built-in absolute timer event has the root element `timer:datetime` and children to specify the time and data of the timer event. For example, the following query would set off a timer at 14:35h on July 31, 2008:

```
event i: timer:datetime {{
  time { "14:35" },
  data { "2008-07-31" }
}}
```

Queries for absolute timer events can also specify periodic timers. For example, the following query would set off a timer event at 16:00h (i.e., 4pm) on every day. A string representing the particular date will be bound to the variable D, which would be useful for example to include the data into a message that is generated from this event query.

```
event i: timer:datetime {{
  time { "16:00" },
  data { var D }
}}
```

Issues related to time and calendars are often difficult since they might be specific to culture or application domain. For example, a stock market application might have the concept of a trading day, which would be the time interval from 9am to 5pm on every day except Saturdays, Sundays, and local bank holidays. XChange^{EQ} does not aim at providing a complete solution to these issues within the event query language. Rather it allows to embed external calendars. These might be specified using a calendar system such as CaTTS [BRS05] or just programmed manually. The external calendar system basically has to generate an appropriate stream of messages from the timer specification. For example, the following event query could set off a timer event for trading days that might be specified in CaTTS. The variable D is bound to the day of the week.

```
catts:tradingDay {{
  dayOfWeek { var D }
}}
```

Built-in timer events The built-in timer of XChange^{EQ} is modeled after the `Calender` class of Java [Sun06]; objects of this class represent a specific date.[2] Figure 6.2 shows an example of the virtual message for the timer at 14:35h on July 31, 2008 in Central European Summer Time (CEST).

[2]The class name "`Calender`" in Java is arguably a bit of a misnomer since its objects represent specific dates according to a calendar system (e.g., Gregorian) and time locales (e.g., time zone), not a calendar system in itself. However, by the time the `Calender` class has been introduced in Java, there already was a `Date` class, which represents dates according to Coordinated Universal Time (UTC).

```
timer:datetime {
    year { 2008 },
    month { "July" },
    day-of-month { 31 },
    hour-of-day { 14 },
    minute { 35 },
    second { 0 },
    millisecond { 0 },

    week-of-year { 31 },
    week-of-month { 5 },
    day-of-year { 213 },
    day-of-week { "Thursday" },
    day-of-week-in-month { 5 },
    am-pm { "pm" },
    hour { 2 },
    zone-offset { 1 },
    dst-offset { 1 },

    date { "2008-07-31" },
    time { "14:35" },
}
```

Figure 6.2: Example of built-in timer message for 14:35h on July 31, 2008 CEST

The multitude of fields in the timer message is necessary to allow for flexible specifications of periodic timer events. For example a query that sets off a timer at 9:00h on every second Tuesday of the month might be specified as:

```
event i: timer:datetime {{
  day-of-week { "Tuesday" },
  day-of-week-in-month { 2 },
  time { "9:00" }
}}
```

It is possible to write timer event queries that do not make sense because they specify impossible dates. In this case, an XChange^{EQ} implementation might refuse to run programs containing such queries or issue a warning. The following are two examples of such timer queries, because there is no February 29, 2100 and because December 24, 2008 is a Wednesday not a Thursday.

```
event i: timer:datetime {{
  date { "2100-02-29" },
  time { "12:00" }
}}
```

```
event i: timer:datetime {{
  data { "2008-12-24" },
  day-of-week { "Thursday" }
}}
```

External calendars Absolute timer events that are defined in external calendar systems are recognized by XChange^{EQ} according to the namespace of the root element. In our `catts:trading-day` example from earlier, the namespace prefix `catts` has to be bound to a namespace URI that is associated in the XChange^{EQ} implementation with an external calendar.

External calendars must generate a sequence of messages together with associated occurrence times from the timer specification (i.e., the query term). Because the sequence may be infinite, it must be provided to XChange^{EQ} in an iterator style interface. The order of the iteration must be ascending according to the ending time. External calendar can thus be understood as a function $c : \mathbb{N} \rightarrow M \times \mathbb{T} \times \mathbb{T}$ (M the set of all data terms/event messages, $\mathbb{T} \times \mathbb{T}$ the set of all time intervals) with $end(t(i) \leq end(t(i+1))$ for all $i \in \mathbb{N}$. To eliminate possibilities of race conditions, XChange^{EQ} is then responsible for actually generating events out of the messages (i.e., initiating a query evaluation step at the ending time associated with the timer message).

6.4 Deductive Rules for Events

Deductive rules are an appropriate way of capturing knowledge about events. They allow to define new, "virtual" events from the base events that are received in the incoming event stream. These new derived events can then in turn be again queried in other rules. For example, we might want to give an explicit representation to big buy events from earlier. This can conveniently be done with a deductive rule.

Deductive rules follow the syntax `DETECT` *event construction* `ON` *event query* `END`. Sometimes we will abbreviate this with "*event construction* ← *event query*"; note however that this is only for notational convenience when giving semantics in this thesis and not core syntax. For the event construction in the rule head, XChange^{EQ} uses the construct terms from Xcerpt (cf. Chapter 4.1.4).

Example The following deductive rule on the left derives a big buy event from buy events using the query from earlier. The derived event contains the identifier of the buy trade, the customer name, and the stock. To its right, we show the new big buy event that would be constructed by this rule from the event in Figure 6.1(d).

```
DETECT
  bigbuy {
    tradeId { var I },
    customer { var C },
    stock { var S }
  }
ON
  event b: buy {{
    tradeId { var I },
    customer { var C },
    stock { var S },
    price { var P },
    volume { var V }
  }}
  where { var P * var V >= 10000 }
END
```

```
bigbuy {
  tradeId { 4242 },
  customer { "John" },
  stock { "IBM" }
}
```

Use of such deductive rules in event query programs makes code more readable and easier to maintain. The rule head gives already a first indication of the intention of the query in the rule body (detecting "big buy" events). Rules processing only big buy events (e.g., to send notifications to an auditor or to aggregate data from big buy events) will not repeat the query in the rule body but can directly work with the big buy event. Should the definition of big buy events change, e.g., due to new legislation to buy events with a total of $20 000 or more, then the necessary change in the code is only local in one rule instead of many rules.

Occurrence time The occurrence time of events that are derived with deductive rules is determined by the events that are used in answering the query in the rule body. If the rule body contains only a single simple event query like here, then the new event simply inherits the occurrence time of the simple event. If the rule body contains an actual complex event query then, as we will see in Section 6.6, the occurrence time of the new event is the time interval covering all events used in answering the complex event query.

Rules for schema mediation In addition to representing inferred event knowledge such as big buy events, rules are also particularly useful for mediating between different event schemas. Because CEP is often used in environments involving many heterogenous applications and event sources, this is a common requirement. For example, our stock market application might receive also orders that follow a different schema than the example in Figure 6.1(d).

Some exemplary variations in the schema are shown in Figure 6.3. In Figure 6.3(a), the event message has the same general structure but different names (in German language) are used for the individual elements. In Figure 6.3(b), the structure of the event message has been changed more fundamentally. In Figure 6.3(c), the order provides a price limit on the total not on the price of a

```
bestellung [
  nummer  { 4711 },
  kunde { "John" },
  kaufen [
    aktie { "IBM" },
    limit { 3.14 },
    anzahl { 4000 }
  ]
]
```

(a) Variation on naming

```
buyorder-ibm [
  id  { 4711 },
  for { "John" },
  maxprice { 3.14 },
  quantity { 4000 }
]
```

(b) Variation in structure

```
buyorder [
  orderId  { 4711 },
  customer { "John" },
  stock { "IBM" },
  volume { 4000 },
  maxspending { 12560 }
]
```

(c) Variation in meaning

Figure 6.3: Example event messages for the stock market application

single stock. The following three deductive rules mediate the schemas by translating all variations into the common schema that has been used so far. The `div` function in the head of the third rule computes the price limit of a single stock as the division of the total price limit by the number of stocks ordered.

```
DETECT
  order [
    orderId { var I },
    customer { var C },
    buy [
      stock { var S },
      limit { var L },
      volume { var V }
    ]
  ]
ON
  event b: bestellung [
    nummer { var I },
    kunde { var C },
    kaufen [
      aktie { var S },
      limit { var L },
      anzahl { Var V }
    ]
  ]
END
```

```
DETECT
  order [
    orderId { var I },
    customer { var C },
    buy [
      stock { "IBM" },
      limit { var L },
      volume { var V }
    ]
  ]
ON
  event b: buyorder-ibm [
    id  { var I },
    for { var C },
    maxprice { var L },
    quantity { var V }
  ]
END
```

```
DETECT
  order [
    orderId { var I },
    customer { var C },
    buy [
      stock { var S },
      limit { div(var M, var V) },
      volume { var V }
    ]
  ]
ON
  event b: buyorder [
    orderId  { var I },
    customer { var C },
    stock { var S },
    volume { var V },
    maxspending { var M }
  ]
END
```

More variations in schema are easily conceivable. For example a variation on the order event might also contain multiple orders for buying or selling different stocks that must be split up into individual buy or sell events. It is interesting to note that this schema mediation already shows a clear benefit of using an event query language with deductive rules like XChange^{EQ} in our application — even though we have no been using actual complex event queries so far.

6.5 Reactive Rules for Events

Deductive rules only derive new events; they do no cause any side-effects. In specifying the event logic of an event-driven system, programmers are concerned with issues of representing knowledge about events and thus a side-effect-free formalism is well-suited. On a large scale however, the eventual goal of event-driven, reactive systems however is to preform actions that have side-effects as response to events. Therefore, it must also be possible to specify reactions to events not just to derive new events.

Messages to external services XChange^{EQ} offers reactive rules for specifying a reaction to the occurrence of an event. The event can be simple or complex, and it can be a base event that is part of the incoming event stream or a derived event that has been generated by some deductive rule.

A typical reaction to an event is to construct a new event message —as with deductive rules— and use the message to call some external service, e.g., a Web Service, an external program, or some procedure. Reactive rules that call to an external service have the form `RAISE` *recipient and message* `ON` *event query* `END`. The following reactive rule is used to simply "forward" big buy events to a Web service located at `http://auditor.com` using SOAP's HTTP transport binding. In the rule head `to()` is used to specify the recipient of the message and the transport method.

```
RAISE
  to(recipient="http://auditor.com",
     transport="http://www.w3.org/2003/05/soap/bindings/HTTP")
  {
    var B
  }
ON
  event b: var B -> bigbuy {{ }}
END
```

Transport binding Other transport bindings (e.g., SOAP over SMTP) could be specified with their URI. XChange^{EQ} further offers the "transport methods" `console`, `java`, and `command`. If `console` is specified, the event message will just be printed on the console, which is mainly useful for debugging of XChange^{EQ} programs. If `java` is specified, a static Java method is called with the event message as parameter. The method must be specified as `recipient` in the form *QualifiedClassName.MethodName* and accept a single parameter of type `org.w3.dom.Document`. If `command` is specified, an external command or program is executed. The command must be specified as `recipient`. The event message as passed to the program on standard input (stdin).

Integration with XChange For tasks involving accessing and updating persistent Web data, XChange^{EQ} event queries can be used together with XChange. XChange Event-Condition-Action rules can then use XChange^{EQ} event queries in their event part. For example, the following XChange rule would maintain a log of buy orders for each customer in a customer database `customers.xml`. Again these buy orders might be generated by deductive XChange^{EQ} rules such as the schema mediation rules from the previous section.

```
ON
  event o: order {{
    customer { var C },
    var B -> buy [[ ]]
  }}
DO
  in { resource { "file:customers.xml" },
    orders {{
      customer {
        name { var C }
        orders {
          insert var B
        }
      }
    }}
  }
END
```

Reactive vs. deductive rules In principle, it is possible to abuse reactive rules to simulate the effect of deductive rules by specifying as recipient the event processing engine itself. As argued in Chapter 5.4, however, this is undesirable. Events generated by reactive rules are subject to timing issues (i.e., they arrive after their cause in the event stream) and have no duration (i.e., even when they are complex events they happen at time points instead of intervals), are misleading to programmers and hinder maintainability of programs, and are less efficient in the evaluation.

6.6 Composition of Events

So far, we have only been looking at queries to single events; we now finally turn to complex events. Complex event queries are written in a style that is reminiscent of logical formulas. XChange^{EQ} has only two operators to compose event queries into such formulas: conjunction and disjunction. Note that this is in contrast to composition-operator-based languages (cf. Chapter 3.2), which offer and require a multitude of composition operators. Due to the separation of concerns in XChange^{EQ}, which treats temporal conditions and event accumulation separately from event composition, two operators suffice in XChange^{EQ}.

Both disjunction and conjunction are multi-ary, allowing to compose any number (≥ 2) of event queries without need for nesting. They are written in prefix notation. Since the simple event queries usually extend over several lines of code this is considered to be more readable than an infix notation. Conjunction uses the keyword `and`, disjunction the keyword `or`. For notational convenience in the semantics, we will also use the infix symbols "$\wedge$" and "$\vee$" instead. Expressions build with `or` and `and` can be nested.

Conjunction When two event queries are composed with `and`, an answer to the composite event query is generated for every pair of answers to the constituent queries. If the constituent queries share free variables, only pairs with "compatible" variable bindings are considered. The following rule illustrates the use of the `and` operator. A "buy order fulfilled" event is detected for every corresponding pair of buy order and buy events. These events have to agree on variable O (the `orderId`). The occurrence time of the detected buy order fulfilled event is the time interval enclosing the respective constituent events.

```
DETECT
  buyorderfulfilled {
    orderId { var O },
    tradeId { var I },
    stock { var S }
  }
ON
  and {
    event o: order  {
      orderId { var O },
      buy {{
        stock { var S }
      }}
    },
    event b: buy {{
      orderId { var O },
      tradeId { var I }
    }}
  }
END
```

This composition of two events queries generalizes into compositions of three of more event queries in the obvious manner.

Disjunction When event queries are composed with `or`, every answer to one of the constituent queries is also an answer to the composite query. The following rule gives an example: every time it recognizes a buy or sell event, it generates a new event signaling the fees (1% of the total) the customer has to pay for the transaction.

```
DETECT
  fees {
    customer { var C },
    amount { mult(0.01, var P, var V) }
  }
ON
  or {
    buy {{
      customer { var C },
        price { var P },
        volume { var V }
    }},
    sell {{
      customer { var C },
        price { var P },
        volume { var V }
    }}
  }
END
```

Disjunctions are not strictly necessary since (in contrast to conjunction) they do not increase expressive power. A rule with a disjunction can be written as multiple rules: Instead of one rule

```
DETECT
  C
ON
  or {
    Q1, Q2, ... Qn
  }
END
```

we can simple write the following n rules

```
DETECT      DETECT            DETECT
  C           C         ...     C
ON          ON                ON
  Q1          Q2                Qn
END         END               END
```

However, disjunction is a considerable convenience in practical programming since it avoids repeating the same rule head multiple times.

6.7 Relative Timer Events

Conjunction of events gives rise to defining relative timer events. Like absolute timer events, relative timer events are not represented as messages in the incoming event stream but must be generated by the query engine “on demand.” The occurrence time of a relative timer event is defined depending on some other event of the same event query. This other event is referred to by its event identifier (`event:` i) in the definition of the relative timer event.

Example The following rule shows a complex event query that queries for an order event and a relative timer event covering the whole time interval between the order event and one hour after the order event.

```
DETECT
  delayedorder {
    var O
  }
ON
  and {
    event o: order {{
      orderId { var O }
    }},
    event t: datetime:extend[event o, 1 hour]
  }
END
```

The effect of the rule is that the event that is generated in the rule head has as occurrence time the time interval stretching from the begin of the order event to one hour after the end of the order event. At present, it might seem that relative timer events are not very useful; essentially the timer event in the example only allows to delay a reaction to order events by one hour if we react to the generated "delayed order" events instead. However, we will see relative timer events become more useful later connection with event accumulation (Section 6.9). With event accumulation we can, for example, query for the absence of a corresponding buy event within the one hour time interval of the timer event so as to detect orders that are "overdue."

Built-In Relative Timers Relative timer events create a new event t with occurrence time $[begin(t), end(t)]$ from another event e with occurrence time $[begin(e), end(e)]$. The following relative timer events are supported natively in XChange^{EQ}. They differ in the occurrence time $[begin(t), end(t)]$ of the created relative timer event t.

- `timer:extend[event` e`,` d`]`: the relative timer event extends the duration of e by a length d at the end, i.e., $begin(t) := begin(e)$, $end(t) := end(e) + d$.
- `timer:shorten[event` e`,` d`]`: the relative timer event shortens the duration of e by a length d at the end, i.e., $begin(t) := begin(e)$, $end(t) := end(e) - d$.
- `timer:extend-begin[event` e`,` d`]`: the relative timer event extends the duration of e by a length d at the begin, i.e., $begin(t) := begin(e) - d$, $end(t) := end(e)$.
- `timer:shorten-begin[event` e`,` d`]`: the relative timer event shortens the duration of e by a length d at the begin, i.e., $begin(t) := begin(e) + d$, $end(t) := end(e)$.
- `timer:shift-forward[event` e`,` d`]`: the relative timer event shifts e forward by length d, i.e., $begin(t) := begin(e) + d$, $end(t) := end(e) + d$.
- `timer:shift-backward[event` e`,` d`]`: the relative timer event shifts e backward by length d, i.e., $begin(t) := begin(e) - d$, $end(t) := end(e) - d$.
- `timer:from-end[event` e`, d]`: the relative timer extends over a length of d starting at the end of e, i.e., $begin(t) := end(e)$, $end(t) := end(e) + d$.
- `timer:from-end-backward[event` e`, d]`: the relative timer extends over a length of d ending at the end of e, i.e., $begin(t) := end(e) - d$, $end(t) := end(e)$.
- `timer:from-start[event` e`, d]`: the relative timer extends over a length of d starting at the start of e, i.e., $begin(t) := begin(e)$, $end(t) := begin(e) + d$.
- `timer:from-start-backward[event` e`, d]`: the relative timer extends over a length of d ending at the start of e, i.e., $begin(t) := begin(e) - d$, $end(t) := begin(e)$.

More complicated computations of timer events can be achieved by defining them relative to other timer events. For example the occurrence time of an event e could be extended by one hour at both the begin and end for a timer event t using a helper timer h as in the following event query:

```
and {
  event e: order {{ }}
  event h: timer:extend[e, 1 hour]
  event t: timer:extend-begin[h, 1 hour]
}
```

The durations d are specified in weeks, days, hours, minutes, seconds, and milliseconds, e.g., `3 weeks 1 day 4 hours 1 min 59 sec 265 ms`. For the keywords `weeks`, `days`, and `hours`, the plural `s` at the end is optional to allow natural expressions when some values are 1.

External Calendars In principle, relative timer events could also be defined using external calendar systems similar to the way XChange^{EQ} allows this for absolute timer events. However, relative timer events have an intimate connection to the garbage collection based on temporal events (cf. Chapter 15). Accordingly, while the language design of XChange^{EQ} is kept extensible for externally defined relative timer events, they are not as easily integrated into the evaluation of event queries as absolute timer events. One possibility of integrating them into the current evaluation method might be to require a conservative approximation for each externally defined relative timer event by one of the built-in timer events. This issue is also discussed further in Chapter 18.

6.8 Temporal (and other) Relationships

Temporal conditions and relationships between events play an important role in querying events. Temporal conditions are specified in the `where`-clause of an XChange^{EQ} event query —just like conditions on event data— and make use of the event identifiers for referring to events.

Example The event query in the following rule involves temporal conditions. It detects situations where a customer first buys stocks and then (re)sells them again at a lower price within a short time, here less than 1 hour. The query illustrates that typical applications require both qualitative conditions (`b before s`) and metric (or quantitative) conditions (`{b,s} within 1 hour`). In addition, the query also includes a data condition for the price (`var P1 > var P2`).

```
DETECT
  earlyResellWithLoss {
    customer { var C },
    stock { var S }
  }
ON
  and {
    event b: buy  {{
      customer { var C },
      stock { var S },
      price { var P1 }
    }},
    event s: sell {{
      customer { var C },
      stock { var S },
      price { var P2 }
    }}
  } where { b before s, {b,s} within 1 hour, var P1 > var P2 }
END
```

As we can see in this example, the approach XChange^{EQ} uses for querying events can be considered as loosely inspired by event calculus. The use of an event identifier `event i:` is somewhat reminiscent of the *happens* predicate found in some variants such as [Kow92] of the event calculus [KS86]. Like an event identifier, the *happens* predicate reifies an event so that conditions on its occurrence time (and other conditions) can be expressed elsewhere in the logical formula. Note however that event calculus and XChange^{EQ} are otherwise very different and serve

different purposes. In Chapter 17 we will give examples why this reification and the separation of the dimension that it achieves is considered better than, e.g., composition operators that include temporal conditions (such a sequence).

In principle, various external calendar and time reasoning systems could be used to specify and evaluate temporal conditions. The declarative semantics of XChange^{EQ} are organized in such a way that embedding an external temporal reasoner is simple. Its operational semantics for the evaluation of event queries however require deeper knowledge about temporal conditions for garbage collection and some optimizations. In the example above, using the condition `b before s` allows (1) to completely avoid evaluating the sell query until a buy event is received and (2) to use the values for variables C and S obtained from buy events when evaluating the sell query. Further, the condition `{b,s} within 1 hour` allows to garbage collect stored buy events after one hour has elapsed. We now discuss first the core temporal conditions supported by the current operational semantics of XChange^{EQ}. We then discuss further temporal conditions that might be desirable but would require at least an extension on the notion of temporal relevance that is used for garbage collection (cf. Chapter 15).

Qualitative Temporal Relations Since the occurrence times of events in XChange^{EQ} are time intervals, Allen's thirteen relations [All83] are the natural candidate to express qualitative temporal conditions. XChange^{EQ} supports all thirteen relations, however they are not of equal importance for querying events. Of primary interest in querying events are:

- i `before` j to express that event i ends before event j starts, i.e., the ending time $end(i)$ of i is lower than the beginning time $begin(j)$ of j: $end(i) < begin(j)$,
- i `contains` j to express that the time interval of event i contains the time interval of event j, i.e., $begin(j) < begin(i)$ and $end(i) < begin(j)$.
- i `overlaps` j to express that event i starts before event j and overlaps j, i.e., $begin(i) < begin(j)$ and $begin(j) < end(i)$ and $end(i) < end(j)$.

Their inverses are available for convenience and named `after`, `during`, and `overlapped-by`, respectively.

The other seven temporal relations can be considered less important since they require at least two endpoints of the time intervals to be exactly equal. Generally in event processing, two different events (or their begins or ends) rarely happen at the exactly same time and if so only accidentally. Nonetheless, XChange^{EQ} supports these relations:

- i `meets` j expresses that the end time of i is the same as the beginning time of j: $end(i) = begin(j)$.
- i `starts` j expresses that i and j begin at the same time and i ends earlier: $begin(i) = begin(j)$ and $end(i) < end(j)$.
- i `finishes` j expresses that i and j end at the same time and i starts later: $end(i) = end(j)$ and $begin(i) > begin(j)$.
- i `equals` j expresses that the time intervals for i and j are exactly equal: $begin(i) = begin(j)$ and $end(i) = end(j)$.

The respective inverses of the first three relations are named `met-by`, `started-by`, and `finished-by` in XChange^{EQ}. The inverse of `equals` is of course again `equals` itself.

Metric Temporal Relations Qualitative conditions express only conditions that constrain the relative positions of occurrence time intervals on the time axis. Complex event queries however also often require conditions that affect the duration of or between events. Since these conditions involve actual values (lengths of time), they are called metric or quantitative.

The metric condition $\{i_1$, $\ldots i_n$ } `within` d ($i_1, \ldots i_n$ event identifiers, $n \geq 0$, d a duration) limits the overall duration of a set $i_1, \ldots i_n$ of events. All events must "fit" into a time interval of length d. More formally: $\max\{end(i_1), \ldots end(i_n)\} - \min\{begin(i_1), \ldots end(i_n)\} \leq d$. The duration d is specified as described previously in Section 6.7. This metric condition is particularly useful for garbage collecting events (cf. Chapter 15)

XChange^{EQ} also supports a constraint that requires a given duration to separate two events. It is written $\{i$, $j\}$ d `apart` (i, j event identifiers, d a duration). Formally it means that either $end(i) - begin(j) \geq d$ or $end(j) - begin(i) \geq d$.

Further Temporal Conditions XChange^{EQ} can easily be extended to accommodate further temporal conditions. In principle, various external calendar and time reasoning systems could also be included. The following gives a flavor of conditions that might be desirable in XChange^{EQ} in addition to the currently supported conditions discussed above.

Qualitative conditions may also refer to a fixed time point or interval on the time axis instead of an event identifier. For example, the following event query would detect only sell events that happen before 9am on September 18, 2006:

```
event s: sell {{ }}
where {
  s before datetime(
    "2006-09-18T09:00")
}
```

Note that there is an important difference between timer events used in queries and references to time as part of `where`-conditions. Timer events have to happen for the event query to yield an answer i.e., they are waited for. Time references in conditions can be in the future and only restrict the possible answers to an event query. The following two event queries are therefore fundamentally different. To illustrate this, consider the following query where the date 9am on September 18, 2006 is used as a timer event:

```
and {
  event s: sell {{ }},
  event t: timer:datetime {{
    date { "2006-09-18" },
    time { "9:00" }
  }}
} where { s before t }
```

All answers to this event query on the are detected at the same time (`2006-09-18T09:00`). In contrast, the answers to the event query further up occur at different times (whenever a sell event is received).

Interesting in temporal conditions are in particular also periodic time intervals. For example an application might have a variant of the early resell complex event from earlier where the buy and sell events must happen on the same trading day (instead of within one hour). This might be specified as follows using an external calendar system such as CaTTS to define the concept of trading day:

```
DETECT
  sameDayResellWithLoss {
    customer { var C },
    stock { var S }
  }
ON
  and {
    event b: buy  {{
      customer { var C },
      stock { var S },
      price { var P1 }
    }},
    event s: sell {{
      customer { var C },
      stock { var S },
      price { var P2 }
    }}
  } where { {b,s} in catts:tradingDay() , b before s, var P1 > var P2 }
END
```

Other Event Relationships The language design of XChange^{EQ} allows easy extension to accommodate other conditions on events such as causal or spatial relationships. This is the power of the separation of the four dimensions of XChange^{EQ}. One could for example have conditions such as `i causes j` for a causal relationship or `i colocated-with j` for a spatial relationship. These relationships have been discussed in Chapter 5.

6.9 Event Accumulation

Event querying displays its differences to traditional querying most perspicuously in non-monotonic query features such as negation or aggregation. For traditional database queries, the data to be considered for negation or aggregation is readily available in the database and this database is *finite*.[3] In contrast, events are received over time in an event stream which is unbounded, i.e., potentially infinite. Applying negation or aggregation on such a (temporally) infinite event stream would imply that one has to wait "forever" for an answer because events received at a later time might always change the current answer.

Therefore, XChange^{EQ} must provide a way to restrict the event stream to a finite temporal extent (i.e., a finite time interval) and apply negation and aggregation only to the events collected in this accumulation window.[4]

It should be possible to determine this accumulation window dynamically depending on the event stream received so far. Typical cases of such accumulation windows are: "from event a until event b," "one minute until event b," "from event a for one minute," and (since events can occur over time intervals, not just time points) "while event c." The last case subsumes the first three, since the they can be defined complex events. Since XChange^{EQ} aims for simplicity with only few language constructs, it supports the last case.

Negation Negation is supported by applying the `not` operator to an event query. The window is specified with the keyword `while` and the event identifier of the event defining the window. The meaning is as one might expect: the negated event query `while` t`: not` q is successful if no event satisfying q occurs during the time interval given by t. The following example detects buy orders that are overdue, i.e., where no matching buy transaction has taken place within one hour after placing the order.

[3]Recursive rules or views may allow to define infinite databases intensionally. However, the extensional data (the "base facts") is still finite.

[4]Keep in mind that accumulation here refers to the way we specify queries, not the way evaluation is actually performed. Keeping all events in the accumulation windows in memory is generally neither desirable nor necessary for query evaluation.

```
DETECT
  buyOrderOverdue {
    orderId { var I }
  }
ON
  and {
    event o: order {{
      orderId { var I },
      buy {{ }}
    }},
    event t: timer:extend[event o, 1 hour],
    while t: not buy {
       orderId { var I }
    }
  }
END
```

The accumulation window is specified by the event query t, which is the relative timer event we have seen earlier in Section 6.7. Observe that the negated query can contain variables that are also used outside the negation, here variable I. The example reveals the strong need to support this.

Aggregation Following the design of the embedded query language Xcerpt, aggregation constructs are used in the head of a rule, since they are related to the construction of new data. The task of the event query in the body is only *collecting* the necessary data or events. Collecting events in the body of a rule is similar to negation and indicated by the keyword `collect` instead of `not`. The following reactive rule has an event query collecting sell events over a full *trading day.*

```
RAISE
  to(recipient="http://example.com",
     transport="http://www.w3.org/2003/05/soap/bindings/HTTP")
  {
    reportOfDailyAverages {
      all entry {
        stock { var S },
        avgPrice { avg(all var P) }
      } group-by var S
    }
  }
ON
  and {
    event t: catts:tradingDay{{ }},
    while t: collect sell {
      stock { var S },
      price { var P }
    }
  }
END
```

The actual aggregation takes place in the head of the rule, where all sales prices (P) for the same stock (S) are averaged and a report containing one entry for each stock is generated. The report is sent at the end of each trading day; this is reflected in the syntax by the fact that `catts:tradingDay{{ }}` must be written as an event, i.e., must actually occur.

Aggregation follows the syntax and semantics of Xcerpt, see Chapter 4.1 for an introduction and [Sch04] for a full account. This shows again that it is beneficial to base an event query language on a data query language. The keyword `all` indicates a structural aggregation, generating an `entry` element for each distinct value of the variable S (indicated with `group-by`). Inside the `entry`-element an aggregation function `avg` is used to compute the average price for each individual stock.

6.10 Stratification: Limits on Recursion

A common problem in deductive rule languages is that recursion and negation make it possible to write programs that do not make sense to the human intuition and are semantically problematic.

The standard example for regular (non-event) rule languages is a program with two rules $p \leftarrow \neg q$ and $q \leftarrow \neg p$. It is possible to write a similar example in XChange^{EQ}:

```
DETECT
  p [ ]
ON
  and {
    event s: s [ ],
    while s: not q [ ]
  }
END
```

```
DETECT
  q [ ]
ON
  and {
    event s: s [ ] ,
    while s: not p [ ]
  }
END
```

In case the incoming event stream contains an event `s[]`^{t} it is questionable if the events `p[]`^{t} and `q[]`^{t} should be derived or not.

To avoid such programs, XChange^{EQ} restricts the use of recursion in rules to so-called statifiable programs. A formal definition is given in Chapter 10. More liberal approaches that have been investigated for Xcerpt such as [BS03] and [Est08] are also be applicable to XChange^{EQ} but not investigated here further.

6.11 Rel^{EQ}: A Simplified, Relational Variant

Queries to XML event messages are highly important in practice and therefore a cornerstone of XChange^{EQ}. However a disadvantage of this is that event queries are rather verbose. Especially for formal and theoretical investigations this verbosity is considered a bit disturbing. Further investigations that relate only to complex events would alway have to carry the full weight of the XML data model and query semantics of Xcerpt. We therefore introduce also a simplified variant called Rel^{EQ}, which also uses a more compact syntax.

The difference between Rel^{EQ} and XChange^{EQ} is that events in Rel^{EQ} are just simple relational facts instead of XML messages. We will however see that Rel^{EQ} preserves all the essentials of XChange^{EQ} and the step of adding XML queries between the two is a small step.

An event in Rel^{EQ} is a simple relational fact together with an occurrence time interval t. It has a predicate name, which corresponds to the event type, and zero or more parameters. For example, the buy event given in XML in Figure 6.1(c) might be expressed as a relational fact of the form `buy(4711, 4242, "John", "IBM", 2.71, 4000)` (analogous for sell events). The XChange^{EQ} "early resell with loss" rule from Section 6.8 would be expressed in Rel^{EQ} as follows:

$$\begin{aligned} \text{loss}(c,s) \quad \leftarrow \quad & b : \text{buy}(i1, t1, c, s, p1, v1),\ \ s : \text{sell}(i2, t2, c, s, p2, v2), \\ & b \ \textsf{before}\ s,\ \ \{b, s\}\ \textsf{within}\ 1h,\ \ p1 > p2; \end{aligned}$$

As we can see, rules use simply a left arrow (*head* ← *body*;) instead of the lengthier `DETECT` *head* `ON` *body* `END`. The keyword `event` before event identifiers is skipped, as is the keyword `var` before variables. The rule body is implicitly surrounded by an `and` conjunction. Disjunction is not supported since its effect can be obtained through writing several rules. Conditions of the `where` clause are written as part of the conjunction. The time units `second`, `minute`, `hour` are abbreviated as s, m, h, or left out altogether is all queries use the same time unit.

The keywords `while`, `not`, `collect` for event accumulation are still used in Rel^{EQ}. Relative temporal events are written with round parentheses instead of square brackets to homogenize syntax with events, e.g., extend$(o, 1h)$. The following rule would detect overdue orders in the same way as the first XChange^{EQ} rule from Section 6.9.

$$\text{overdue}(i) \leftarrow o : \text{order}(i, c1, s1, l, v1),\ \ t : \text{extend}(o, 1h),\ \ \textsf{while}\ t : \textsf{not}\ \text{buy}(i, t, c2, s2, p, v2);$$

Note that the simple event queries in Rel^{EQ} have no sense of incompleteness as the Xcerpt queries used in XChange^{EQ}. Therefore Rel^{EQ} queries must use some variables that would not be needed in XChange^{EQ} queries. This includes $i1$, $i2$, $v1$, $v2$, and others in the above examples.

For aggregation, Rel^{EQ} supports the aggregation functions `min`, `max`, `sum`, `count`, and `avg`. Since the data model are flat relational facts, they cannot be nested and no structural aggregation like building of lists is supported. There is no explicit grouping (like the `group by` clause); instead the other non-aggregated variables in the rule head are used for that. The following rule reports the number of orders that have been placed in the last hour whenever an overdue event happens:

$$\begin{array}{rcl} \text{report}(\text{count}(i)) & \leftarrow & o : \text{overdue}(i1),\ t : \textsf{extend-begin}(o, 1h), \\ & & \textsf{while}\ t : \textsf{collect}\ \text{order}(i, c, s, l, v); \end{array}$$

To count the number of orders in the last hour before an overdue event *on a per stock basis*, the rule could be modified to:

$$\begin{array}{rcl} \text{report-per-stock}(\text{count}(i), s) & \leftarrow & o : \text{overdue}(i1),\ t : \textsf{extend-begin}(o, 1h), \\ & & \textsf{while}\ t : \textsf{collect}\ \text{order}(i, c, s, l, v); \end{array}$$

Because variable s is now in the head, different `report-per-stock` events are generated for each distinct value of s.

Chapter 7

Use Cases

To deepen our informal understanding of XChange^{EQ} and to see it in action, this chapter describes and implements a number of practical use cases. The use cases aim to be illustrative for XChange^{EQ} and thus only show the event logic in a number of possible application scenarios without worrying about the full application logic. The previous chapter has already given a flavor of a use case with its stock market application. In this chapter, we consider two further use cases, monitoring the business activities in an order processing application (Section 7.1) and working with sensor events in a SCADA Application (Section 7.2).

7.1 Business Activity Monitoring in Order Processing

Our first use cases is concerned with monitoring of events that signify business activities in an order processing application. Complex events in this application include for example the completion of an order processing or reports of the daily and weekly activities.

7.1.1 Mediating Order Events

Orders in our application might be placed in a number of different ways. We use rules to mediate these different events that signify orders into a single common type of order event. We first consider event that signify a single order but use different XML tags (here: in German language) than our desired mediated order event. An example of the incoming order event is shown on the left, the rule on the right "translates" this event into the mediated order event.

```
bestellung {
  nummer { 4711 },
  kunde { "John" },
  produkt { "Muffins" },
  anzahl { 23 }
}
```

```
DETECT
  order {
    id { var I },
    customer { var C },
    product { var P},
    quantity { var Q }
  }
ON
  event b: bestellung {
    nummer { var I },
    kunde { var C },
    produkt { var P },
    anzahl { var Q }
  }
END
```

As we can see from the rule head, orders in our application have the following event data associated with them: an `id` to uniquely identify our order, the name of the `customer` that places the order, the ordered `product`, and the `quantity` that is ordered.

An order might also be a complex event. For example, an order might come about by an offer that a customer chooses to accept. The following rule on the right detects an order event as

composition of a weekly offer event and an acceptance of the offer. Examples for these two events are shown on the left. The offer, and thus the resulting order event, include a `discount`.

```
weekly-offer {
  code { 42 },
  product { "Muffins" },
  discount { 10 },
  max-quantity { 50 }
}
```

```
accept-offer {
  code { 42 },
  orderId { 4711 },
  customer { "John" },
  quantity { 29 }
}
```

```
DETECT
  order {
    id { var I },
    customer { var C },
    product { var P},
    quantity { var Q },
    discount { var D }
  }
ON
  and {
    event o: weekly-offer {
      code { var K },
      product { var P },
      discount { var D },
      max-quantity { var MQ }
    },
    event a: accept-offer {
      code { var K },
      id { var I },
      customer { var C },
      quantity { var Q }
    }
  } where { o before a, {o,a} within 1 week, var Q <= var MQ }
END
```

One might argue that the order event might simply be detected through the acceptance of the offer without composing this with the preceding weekly offer. This is however not sound: First, note that the acceptance event misses certain data that is part of the resulting order event. This includes the ordered `product` and the `discount`; both must be extracted from the weekly offer. Second, weekly offers are only valid for a given time, here one week. This constraint is expressed in the `where`-clause of the event query. Third, weekly offers also specify a maximal quantity (`max-quantity`) that may be ordered for the discounted price. The event query checks that the ordered quantity does not exceed this maximum. Finally, the composition of offer and acceptance ensures that customers cannot "cheat" and accept offers that do not actually exist or prematurely accept offers (i.e., send an acceptance before an offer).

Our order events contain only a single product. However, a customer might place multiple orders with a single event, e.g., by going to checkout with a shopping basket including several products on a Web site. A possible example of such an event that includes multiple orders is shown on the left. The rule on the right splits these orders into separate order events as desired for our order processing application.

```
multi-order {
  customer { "John" },

  item {
    id { 4711 },
    product { "Muffins" },
    quantity { 29 }
  },

  item {
    id { 4712 },
    product { "Coffee" },
    quantity { 2 }
  }
}
```

```
DETECT
  order {
    id { var I },
    customer { var C },
    product { var P },
    quantity { var Q }
  }
ON
  event m: multi-order {{
    customer { var C },

    item {
      id { var I },
      product { var P },
      quantity { var Q }
    }
  }}
END
```

Note that all incoming events include already identifiers for the orders, which become the `id` elements in the resulting order events. In some applications, such an identifier should rather be generated by the event logic than expected in the input events. However, generation of such

identifiers poses some problems, in particular for declarative semantics. We discuss this issue in Chapter 18.3.4.

7.1.2 Completed Orders

In our application, we might want to detect situations when the processing of a given order has been completed. “Completed” here means that the ordered products have been delivered to the customer. A possible reaction to such a completion event then might be to initiate the billing process; we will also see the completion event be used later for counting the number of completed orders on a per-day basis.

Our complex event query to detect such completed orders involves the composition of three events: the original order event, the shipping event that is generated when the package is dispatched to a shipping company, and the tracking event that signals the successful delivery of the package. Examples of these events are shown on the left, the rule detecting the completion event on the right.

```
order{
  id { 4711 },
  customer { "John" },
  product { "Muffins" },
  quantity { 29 },
  discount { 10 }
}
```

```
shipping {
  orderId { 4711 },
  trackingNumber { 4242 }
}
```

```
tracking {
  trackingNumber { 4242 },
  status { "delivered" }
}
```

```
DETECT
  completion {
    id { var I },
    product { var P }
  }
ON
  and {
    event o: order {{
      id { var I },
      product { var P }
    }},
    event s: shipping {
      orderId { var I },
      trackingNumber { var T }
    },
    event d: tracking {{
      trackingNumber { var I },
      status { "delivered" }
    }}
  }
  where { o before s, s before d,
          {o,d} within 1 week }
END
```

Note that, as before with the offer and acceptance of an offer, a reaction only to the tracking event is not sound. Our application might receive other tracking events that are not related to orders, e.g., for shipments of replacements or free catalogs. Further, the tracking events does not contain all necessary information needed to generate the completion event. The shipping event in the middle is needed in order to connect together the order event, which has only an order id, and the tracking event, which only has a tracking number. The query also includes a metric temporal constraint `{o,d} within 1 week`. Even if not necessary for the event logic as such, this constraint is necessary to enable garbage collection of events. Other forms of garbage collection that might not require such a temporal constraint are discussed in Chapter 15 and 18. The time frame of one week here is chosen conservatively so that other events would signal incomplete orders before (see the rules coming just up).

7.1.3 Overdue Orders and Late Deliveries

In our scenario it might be useful to monitor the quality of service of our order processing. For example, we might want to detect orders that have not been shipped within a given time, i.e., that are overdue. The time frame when an order is overdue might however be different depending on whether the order makes use of a discount or how many items were ordered. The following three rules detect overdue orders as follows: orders of less than 10 items without a discount must be shipped within 6 hours, order of less than 10 items with a discount have 12 hours, and orders of 10 or more items have 24 hours.

```
DETECT
  overdue{ var I }
ON
  and {
    event o: order {{
      id { var I },
      quantity { var Q },
      without discount {{ }}
    }}
    event w: timer:extend[
      event o,
      6 hours
    ],
    while w: not shipping {{
      orderId{ var I }
    }}
  } where { var Q < 10 }
END
```

```
DETECT
  overdue{ var I }
ON
  and {
    event o: order {{
      id { var I },
      quantity { var Q },
      discount {{ }}
    }}
    event w: timer:extend[
      event o,
      12 hours
    ],
    while w: not shipping {{
      orderId{ var I }
    }}
  } where { var Q < 10 }
END
```

```
DETECT
  overdue{ var I }
ON
  and {
    event o: order {{
      id { var I },
      quantity { var Q }

    }}
    event w: timer:extend[
      event o,
      24 hours
    ],
    while w: not shipping {{
      orderId{ var I }
    }}
  } where { var Q >= 10 }
END
```

In a similar manner, we might want to monitor the tracking events from our shipping company to detect when it is late with its delivery. The company has 1 day for orders involving less then 10 items and 2 days for larger orders. The following rule uses an `or` nested into the outer `and` to cover both cases within one event query.

```
DETECT
  late[ var I, var T ]
ON
  and {
    event o: order {{
      id { var I },
      quantity { var Q }
    }},
    event s: shipping {
      orderId{ var I },
      trackingID { var T }
    }
    or {
      event w: timer:extend[event s, 1 day]  where { var Q < 10  },
      event w: timer:extend[event s, 2 days] where { var Q >= 10 }
    }
    while w: not tracking {{
      trackingNumber { var I },
      status { "delivered" }
    }}
  }
  where { {o,s} within 1 day }
END
```

Whether a single rule like this or two separate rules are preferable is often a matter of taste. A single rule like this bundles all event logic for late deliveries in one place and is more compact.

7.1.4 Diagnostics for Overdue Orders

When an overdue event such as above occurs, we might want to automatically generate some diagnostics that help us identify potential problems in our order processing. For example we might want to count the number of shipping events that have occurred in the last 24 hours before the overdue order. A high number might indicate that the shipping department is overloaded, a lower number that the problem is elsewhere. The following rule implements this counting of shipping event and reports the result immediately whenever an overdue event happens.

```
DETECT
  shipping-load { count(all var S) }
ON
  and {
    event o: overdue {{ }},
    event w: timer:from-end-backward[event o, 24 hours],
    while w: collect var S -> shipping {{ }}
  }
END
```

Note that in the rule the 24 hour time interval starts with the end of the overdue event.

7.1.5 Daily and Weekly Reports

Monitoring of events is also interesting for generating periodic reports. The following rule on the right generates a daily report, which includes the number of orders on that day as well as the number of completed orders. Over time, these report events might be used to identify trends in the number of orders received or potential upcoming bottlenecks in our order processing. The rule on the left generates an event for every day using the `timer:datetime` events.

```
DETECT
  day [ var WD, var DATE ]
ON
  and {
    event b: timer:datetime {{
      day-of-week { var WD },
      date { var D },
      time { "0:00" }
    }},
    event e: timer:datetime {{
      time { "23:59" }
    }}
  }
  where { b before e, {b,e} within 1 day }
END
```

```
DETECT
  daily-status-report {
    var D,
    orders { count(all var O) },
    completed { count(all var C) }
  }
ON
  and {
    event w: var D -> day [[ ]],
    while w: collect order {{
      id{ var O }
    }},
    while w: collect completion {{
      id{ var C }
    }}
  }
END
```

Instead of using two absolute timer events in the left rule, it would also be possible to use just one absolute timer event and a relative timer event that extends this absolute timer by one day.

We also might want to generate reports of our weekly sales figures (or rather: “order figures”). The following left rule generates a week event for this. The rule on the right generates the weekly report. It list every product that has been ordered together with the number of orders and the total quantity of ordered items of that product.

```
DETECT
  week [ var W, var Y ]
ON
  and {
    event b: timer:datetime {{
      year { var Y },
      week-of-year { var W },
      day-of-week { "Monday" },
      time { "0:00" }
    }}
    event e: timer:datetime {{
      day-of-week { "Sunday" },
      time { "23:59" }
    }}
  }
  where { b before e, {b,e} within 1 week }
END
```

```
DETECT
  weekly-sales-report {
    var W,

    all item {
      var P,
      total-orders { count(all var I) },
      total-quantity { sum(all var Q) }
    } group by { var P }
  }
ON
  and {
    event w: var W -> week {{ }},
    while w: collect order {{
        id { var I },
        var P -> product {{ }}},
        quantity { var Q }
    }}
  }
END
```

Interesting in the right rule is particularly the use of nested grouping and aggregation. It shows a complexity of querying and constructing XML data that is not possible when events are represented only as relational tuples as in some other event query languages.

7.2 Monitoring of Sensor Events in a SCADA Application

Our second use case is concerned with events that are generated by sensors. We consider a larger scale facility (e.g., a train station) in which temperature and smoke sensors are deployed. A SCADA system for such a facility might want to monitor the events generated by these sensors to detect emergencies such as fires but also necessary maintenance work of detect sensors.

We assume that our facility is divided into several areas. These areas may be overlapping so that a single sensor can belong to several areas.

7.2.1 Simple Sensor Failures

Temperature sensors in our scenario periodically generate an event roughly every 10 seconds that contains the currently measured temperature. Smoke sensors only generate events when they actually detect smoke. Additionally, they periodically signal that they are still working, roughly every 60 seconds. The events from both types of sensors also include a serial number identifying the particular sensor generating the event and list the areas in which the sensor is located. The following are examples for a temperature reading event, a smoke event, and an alive event.

```
temperature-signal {
  sensor { "ABCD-1234" },
  areas {
    area { 51 },
    area { 66 }
  },
  temperature { 20 }
}
```

```
smoke-signal {
  sensor { "EFGH-5678" },
  areas {
    area { 51 },
    area { 42 }
  },
  smoke { }
}
```

```
smoke-signal {
  sensor { "EFGH-5678" },
  areas {
    area { 51 },
    area { 42 }
  },
  alive { }
}
```

The simplest kind of sensor failure is when a sensor stops sending events, e.g., because it is disconnected from the network or has a hardware defect. In our scenario we can detect such failures as absences of sensor events. Temperature sensors send a reading roughly every 10 seconds. Therefore if some reading is not followed by another reading within a time span slightly larger than 10 seconds (say, 12 seconds) we can conclude a failure of the temperature sensor. Similarly for the smoke sensors, only here we use alive events and a longer time span of say 65 seconds. The following rules detect failures of temperature and smoke sensors this way:

```
DETECT
  failure {
    sensor-type { "temperature" },
    var S,
    var A,
    last-temperature { var T }
  }
ON
  and {
    event t: temperature-signal {
      var S -> sensor {{ }},
      var A -> areas {{ }},
      temperature { var T }
    },
    event w: timer:extend[event t, 12 sec],
    while w: not temperature-signal {{
      var S
    }}
  }
END
```

```
DETECT
  failure {
    sensor-type { "smoke" },
    var S,
    var A
  }
ON
  and {
    event t: smoke-signal {
      var S -> sensor {{ }},
      var A -> areas {{ }},
      alive {{ }}
    },
    event w: timer:extend[event t, 65 sec],
    while w: not smoke-signal {{
      var S,
      alive {{ }}
    }}
  }
END
```

7.2.2 Cleaning of Sensor Data

Sensor measurements are typically subject to certain errors in measurement as well as to influences from the environment. For example, a smoke sensor might be momentarily set of by dust particles in the air. Similarly, temperature measurements are subject to fluctuations. Complex event queries can be useful for cleaning such sensor data.

Data from temperature sensors can be cleaned effectively by simply averaging measurements over a time interval. Here we use time intervals of one minute lengths that are adjacent and non-overlapping, starting with every full minute of wall clock time. The average is computed individually for each sensor (due to the `group by` on the sensor serial number in variable S).

```
DETECT
  all temperature-avg {
    var S,
    var A,
    average { avg(all var T) }
  } group by { var S }
ON
  and {
    event m: timer:datetime {{
      second { 0 },
      millisecond { 0 }
    }},
    event w: extend[event m, 1 min],
    while w: collect temperature-signal {
      var S -> sensor {{ }},
      var A -> areas {{ }},
      temperature { var T }
    }
  }
END
```

Such an averaging smoothes out outliers in the temperature values where a sensor momentarily measures a far to high or low temperature and then immediately returns to regular measurements.

Our smoke sensors do not supply numeric values or measurements. However they are still susceptible to false alarms. Because, for example, dust particles and the like might set of a smoke sensor, a single smoke event from a sensor should not be considered an alarm. Further, several smoke events within a short period of time might be generated due to the same dust particles. We therefore want to consider if smoke has already been detected in a time interval somewhat before a given smoke event. The following two rules realize this. The first rule counts the number of smoke signal that happen in a 10 second interval not immediately preceding a smoke signal, but happening another 10 seconds before that smoke signal (i.e., in the time interval starting 20 seconds before the signal and lasting until 10 seconds before the signal). This count is used to generate a helper event that is queried by the third rule. If it exceeds three, then an actual alarm is raised.

```
DETECT
  smoke-counter [
    var S,
    var A,
    signals { count(all var C) }
  ]
ON
  and {
    event s: smoke-signal {{
      var S -> sensor {{ }},
      var A -> areas {{ }},
      smoke {{ }}
    }},
    event h: timer:shift-forward[event s, 10 sec],
    event w: timer:extend-begin[event h, 10 sec],
    while w: collect var C -> smoke-signal {{
      var S,
      smoke {{ }}
    }}
  }
END
```

```
DETECT
  smoke-alarm {
    var S,
    var A
  }
ON
  smoke-counter [
    var S,
    var A,
    signals { var C }
  ]
  where { var C >= 3 }
END
```

Note that in XChange^{EQ} two rules must be used to express a condition on aggregated values such as the count of smoke signals here. This is inherited from the underlying Web query language Xcerpt, where also two rules would have to be used. Some query languages would allow to do this more conveniently in one expression, e.g., SQL uses the `HAVING` clause in queries to express conditions on aggregated values. For more convenience, Xcerpt could be easily extended to provide such syntactic sugar and these extensions would be directly applicable to XChange^{EQ}.

7.2.3 Fire Alarm

Both, a single (averaged) high temperature or a single smoke alarm might not reliably indicate an actual fire. For example, a smoke alarm might be go off due to a cigarette smoke or a high temperature might be measured due to direct exposure to sun light. In our scenario, we therefore detect detected fire alarms as a combination of high average temperatures at two different sensors and a smoke alarm. Of course, all sensors must be located in the same area. These events must happen within a short period of time, here 90 seconds. (Note that average temperature events extend over a duration of one minute, therefore durations close to or under 60 seconds would not have the intended result.)

```
DETECT
  fire-alarm {
    area { var A }
  }
ON
  and {
    event t1: temperature-average {{
      sensor { var S1 },
      areas {{
        area { var A }
      }},
      average { var T1 }
    }},
    event t2: temperature-average {{
      sensor { var S2 },
      areas {{
        area { var A }
      }},
      average { var T2 }
    }},
    event s: smoke-alarm {{
      area { var A }
    }}
  }
  where { var T1 > 60, var T2 > 60, var S1 != var S2,
          {t1,t2,s} within 90 seconds }
END
```

Note that the condition `var S1 != var S2` is needed to ensure that different temperature sensors are actually used in detecting the complex event. Without this condition both $t1$ and $t2$ might be bound to the same event from the same sensor.

The logic of this rule for detecting a fire entails that sensors manage to communicate their readings before they are affected by the fire in their area. This might not always be the case, the fire might destroy the communication links or the sensors themselves before. We therefore also might want a rule that raises an alarm if three or more sensors in the same area fail (as detected with earlier rules) within a short time (here again 90 seconds). Because we cannot be sure that the sensor failures are due to fire —they might also be due to flooding or other causes— the following rules raise general alarm rather than a fire alarm.

```
DETECT
  failure-counter {
    area { var A },
    failures { count(all var S) }
  }
ON
  and {
    event f: failure {{
      areas {{
        area { var A }
      }}
    }},
    event w: timer:from-end-backward[event f, 90 sec],
    while w: collect failure {{
      sensor { var S },
      areas {{
        area { var A }
      }}
    }}
  }
END
```

```
DETECT
  general-alarm {
    area { var A }
  }
ON
  event f: failure-counter {
    area { var A },
    failures { var C }
  }
  where { var C >= 3 }
END
```

Note that as before, the condition on the count must be realized through two separate rules.

7.2.4 Remarks on Avoiding Alarm Fatigue

The realization of fire alarm events above might be considered to have a minor shortcoming. It potentially generates many alarms for the same area. For example if there are three sensors reporting high average temperatures, not just two, then the query would generate an alarm for each pair of the three, i.e., three alarms. Similarly, two smoke alarms from different sensors would lead to two fire alarms for the same area. This issue is a common issue also with other event query languages. One potentially dangerous effect of this can be “alarm fatigue,” where users start ignoring alarms or alarms of a certain kind because too many alarms that are not meaningful are generated.

A possibility to avoid such alarm fatigue in our use case might be to avoid the generation of alarms when there already has been an alarm in a given previous time frame. To do this, we need two rules: one detects candidates for fire alarms like the first rule from the previous section; the other generates actual alarms from a alarm candidates but avoids generating such an actual alarm when there has already been some alarm candidate within a time frame of one hour earlier.

```
DETECT
  fire-alarm-candidate {
    area { var A }
  }
ON
  and {
    event t1: temperature-average {{
      sensor { var S1 },
      areas {{
        area { var A }
      }},
      average { var T1 }
    }},
    event t2: temperature-average {{
      sensor { var S2 },
      areas {{
        area { var A }
      }},
      average { var T2 }
    }},
    event s: smoke-alarm {{
      area { var A }
    }}
  }
  where { var T1 > 60, var T2 > 60,
          var S1 != var S2,
          {t1,t2,s} within 90 seconds }
END
```

```
DETECT
  fire-alarm {
    area { var A }
  }
ON
  and {
    event c: fire-alarm-candidate {
      area { var A }
    },
    event h: timer:shift-forward[
               event c, 1 sec ],
    event w: timer:extend-backward[
               event h, 1 hour ],
    while w: not  fire-alarm-candidate {
      area { var A }
    }
  }
END
```

In the second rule we are using a "trick" that is somewhat unsatisfactory by having the time window last only until one second before the candidate alarm not until the alarm candidate. Consider what would happen when we write the second rule incorrectly as follows:

```
DETECT
  fire-alarm {
    area { var A }
  }
ON
  and {
    event c: fire-alarm-candidate {
      area { var A }
    },
    event w: timer:extend-backward[event c, 1 hour],
    while w: not  fire-alarm-candidate {
      area { var A }
    }
  }
END
```

The body of this rule is unsatisfiable, i.e., never can have an answer (taking essentially the form like the logic contradiction $x \wedge \neg x$): any event for the c alarm candidate lies also in the time window of w where we query the absence of such an alarm candidate. Hence, the rule will never derive an actual fire alarm.

One way to avoid using an unsatisfactory trick like in the correct solution above would be to support variants of `while` that collect not over the closed interval of w but (half-)open intervals, i.e., intervals where one of the boundaries or both are not considered part of the interval. In the example when w is open, the candidate alarm event for c will not lie in w and thus the incorrect rule be made to work correctly. We discuss this extension to XChange$^{\text{EQ}}$ in Chapter 18.2.4.

Event instance consumption and selection in composition-operator-based event query languages might provide an alternative solution to working with negation (over a time window) for avoiding alarm fatigue. However they cannot be considered a satisfactory solution.

A further option might be to work with activating and deactivating rules (which is not possible in XChange$^{\text{EQ}}$). This however would make semantics of the query language very involved and state-based (to maintain which rules are active and which are not). Also, such an activation and deactivation might lead to incorrect results: we still want fire alarms for other areas to be generated.

Part III

Declarative Semantics

Chapter 8

Declarative Semantics: Motivation and Overview

Having introduced XChange^{EQ} informally in the preceding chapters, we now supply formal, declarative semantics for XChange^{EQ} in the following chapters. This chapter gives first a motivation for developing declarative semantics (Section 8.1), in particular in contrast to giving only operational semantics. It then outlines the requirements and desiderata for declarative semantics of an event query language such as XChange^{EQ} (Section 8.2). Finally, it gives a brief overview over the general approach used for the declarative semantics of XChange^{EQ} (Section 8.3). The declarative semantics of XChange^{EQ} that are presented in this and the following chapters have also been discussed with less detail in [BE07a] and [BE07c].

8.1 Motivation

Formal and declarative semantics are as desirable for an event query language as they are for "non-event" query languages, i.e., database query languages, Web query languages, etc. Formal, declarative semantics relate the syntax of the language to mathematical objects and expressions that capture the intended meaning. Being grounded in the rigor of mathematics, formal semantics avoid ambiguities that might exist in informal descriptions of the language semantics [McD86]. They thus provide a reference to implementors of the language and help greatly in standardization efforts.

Declarative semantics focus on expressing *what* a sentence in the language means, rather than *how* that sentence might be evaluated. Declarative semantics thus provide a convenient basis to prove the correctness of various operational semantics. In particular in the area of query languages there are usually a myriad of equivalent ways to evaluate a given query, that is, of possible operational semantics. If on the other hand formal semantics of a language were specified only in an operational way, proving the correctness of other operational semantics would be significantly harder: since operational semantics focus on how the result is computed not on what is the result, we have to reason about the equivalence of two computations. When we prove correctness of operational semantics w.r.t. declarative semantics, we instead just reason about properties of the output of one computation. This use of declarative semantics to prove correctness of evaluation methods is particularly useful in research on optimization.

Formal, declarative semantics also give rise to proofs about the event query language in general, certain classes of queries, or individual queries. For query languages, it is for example common to identify classes of queries with complexity classes. In the context of event queries, we might ask if a language actually "makes sense" on event streams. That is, can all queries that can formulated in it be evaluated without waiting for the end of the event stream.

Finally, easy to understand and "mathematically aesthetic" declarative semantics are arguably an indication of a good language design and work towards such declarative semantics might help

identifying incongruities and flaws in the language.

Declarative semantics have often been neglected in event query languages so far (see also Chapter 3). Semantics of composition-operator-based, if specified at all, usually have an operational, "algebraic" flavor. The semantics of a composition operator are often specified as a function between sequences (or histories or traces) of events [HV02, CL04]. When the language includes consumption modes, then the sequence of events must be modified when answers are detected, making the semantics essentially state-based (like those of imperative programming languages). Further, event data is an aspect typically neglected in the semantics of composition operators.

Semantics of data stream query languages are based on the semantics of a relational query language (such as SQL), which typically will provide well-rounded declarative semantics. However, the conversion of streams into relations and back again makes semantics of data stream languages somewhat unnatural and unintuitive. The semantics of the relation-to-stream operators `Istream` and `Dstream` further are defined as the difference between the current and previous state of the result relation. They thus also become to some degree stateful, although not as much as composition operators with consumption.

Semantics of production rules finally are fully state-based and might be argued to be as difficult and undeclarative as semantics of traditional imperative programming languages. This is particularly so when considering that actions in production rules are usually expressed in a host programming language. Further, the semantics of production rule languages include an involved conflict resolution scheme to determine the order in which conflicting rules, i.e., rule that are triggered at the same time, fire.

8.2 Requirements and Desiderata

Our goal with XChange^{EQ} are semantics that are natural on event streams; they should not require a conversion from streams to relations and back. Also, they should be as declarative as possible and thus avoid any notion of state. Since event queries in XChange^{EQ} are affected by data in events, declarative semantics for XChange^{EQ} must put emphasis on this event data that is extracted into variable bindings. XChange^{EQ} supports deductive rules over events that must be accommodated in its declarative semantics.

A recurrent theme in this thesis is to apply and adapt formalisms from traditional, non-event query languages to event query languages. This allows to leverage —sometimes with important changes— many existing results from database theory and logic programming, sheds light on where new concepts are needed for event queries, and finally avoids reinventing the wheel where not necessary.

Query languages with deductive rules such as datalog [AHV95] or Xcerpt [Sch04] specify their semantics with a model theory. Obviously, this approach accounts well for data in events as well as deductive rules. Model theories are considered highly declarative, in particular since they are defined recursively over the structure of a formula, allowing to consider sub-formulas in isolation. Model theories are also theoretically well-understood and relatively easy to understand. Because of their connection with first order logic, model theories are also interesting for extending a rule-based event query language towards more advanced knowledge representation or for specifying the semantics of integrity constraints.

For these reasons and because XChange^{EQ} builds upon Xcerpt, it is desirable to also use a model theoretic approach for specifying declarative semantics of XChange^{EQ}. There are however some important differences between traditional, non-event query languages and an event query language such as XChange^{EQ}. These differences make some adaption of the traditional approach necessary:

- Temporal relations and timer events have a fixed interpretation.
- In addition to normal variables, event queries in XChange^{EQ} use event identifiers. These event identifiers differ from regular variables in that they are bound to events, rather than data terms.

- Events and, importantly, answers to complex event queries have an occurrence time.
- Most importantly, the semantics must be sensible for *infinite streams*.

The last and most important difference between event queries and traditional queries will not inherent to the approach used for giving declarative semantics of XChange$^{\mathrm{EQ}}$. Rather we will have to prove as a theorem that the declarative semantics of XChange$^{\mathrm{EQ}}$ really satisfy this requirement.

8.3 Overview of our Approach

The problem that the declarative semantics of XChange$^{\mathrm{EQ}}$ address can be described on an a very abstract level as follows: given a set of deductive rules (also called program) together with an incoming event stream, i.e., the set of events that happen in the outside world and are not derived by rules, it tells us all events that are derived by the rules. This approach then also covers the issue of finding the answers for a given event query (e.g., that is used in a reactive rule) w.r.t. a program and an incoming event stream.

The idea behind the model theory of XChange$^{\mathrm{EQ}}$ (and other model theories) is view expressions of the language (rules, queries) as sentences of first order logic (or, since events are represented as data terms with occurrence times not logic facts, something similar). These sentences are related to an interpretation by defining an entailment relation. An interpretation contains all events that happen, the base events from the incoming event stream and the derived events. The entailment relation is defined recursively over the structure of the sentences. Of interest for the semantics are those interpretations that (1) satisfy all rules of a program w.r.t. the entailment relation and (2) contain the stream of incoming events. These interpretations are then called models. The model theory of XChange$^{\mathrm{EQ}}$ takes into accounts the special requirements of an event query language as discussed above and will be defined in Chapter 9.

A model theory has the issue of allowing many models for a given program. To give precise semantics to XChange$^{\mathrm{EQ}}$, we need a unique model and this model should be natural and intuitive. A common and convenient way to obtain such a unique model is to define an accompanying fixpoint theory, which is based on the model theory. The fixpoint interpretation specified by the fixpoint theory is the intended model of an XChange$^{\mathrm{EQ}}$ program. When non-monotonic features like negation or aggregation are combined with recursion of rules, this might lead to some issues that have been mentioned in Chapter 6.10. By restricting ourselves to stratifiable programs, we avoid such cases and ensure that a unique fixpoint interpretation exists. A formal definition of stratification is given together with the definition of the fixpoint interpretation in Chapter 10.

We then have to show that the fixpoint interpretation of stratified programs is really well-defined and unambiguous. Further, and more importantly, we have to show that the semantics specified by the model theory and the fixpoint interpretation "make sense" on infinite event streams. It must be possible to evaluate queries in a streaming manner, where answers are generated "online" and we never have to wait for the stream to end. We state all this formally as theorems and give proofs in Chapter 11

Chapter 9

Model Theory

The idea of a model theory, as it is used in traditional, non-event query languages [Llo93, AHV95, Sch04], is to relate expressions to an *interpretation* by defining an *entailment relation.* Expressions are syntactic fragments of the query language such as rule, queries, or facts viewed as logic sentences. The interpretation contains all facts that are considered to be true. The entailment relation indicates whether a given interpretations entails a given sentence from the language, that is, if the sentence is logically true under this interpretation. Of interest for the semantics of a given query program and a set of base facts are then those interpretations that (1) satisfy all rules of the program and (2) contain all base facts. Because it satisfies all rules, such an interpretation contains in particular also all facts that are derived by rules. We call these interpretations *models.*

When we replace facts that are true with events that happen, this approach can also be applied to event query languages. The problem, of course, is that events are associated with *occurrence times* and event queries are evaluated *over time* against a potentially *infinite event stream.* At each time point during the evaluation we know only which events have happened (i.e., been received in the event stream) so far, not any events that might happen in the future.

A core idea in giving model-theoretic semantics to XChange$^{\text{EQ}}$ is to pretend a kind of "omniscience." We assume that we know the full, infinite event stream that contains all events that ever happen together with their occurrence times. The semantics are then specified irrespective of the evaluation times of event queries and rules.

With this approach, it is however not clear that the semantics actually "make sense" for event queries. The semantics might easily imply cases where the streaming evaluation of an event query over time is simply impossible because it might require waiting indefinitely for the end of the event stream or crystal gazing into the future. Of course, XChange$^{\text{EQ}}$ and its semantics have been defined in such a way that a streaming evaluation is possible, and we will prove formally in Chapter 11.

The idea of pretending omniscience in the definition of the semantics of an event query language might seem simple enough. However, it turns out that previous event query languages have not defined their semantics this way (see Chapter 3). The semantics that they end up with can be considered more complicated and, because they are to some degree stateful, less declarative. The approach of XChange$^{\text{EQ}}$ in contrast leads to very intuitive and highly declarative semantics.

9.1 Basic Definitions: Time and Events

We start of with some basic definitions that explain how we represent time and events in the semantics of XChange$^{\text{EQ}}$.

9.1.1 Data Terms

Data terms from Xcerpt are used to represent data and type information for events. Data terms are just syntactic objects. We have seen many examples of data terms in the previous sections. A simplified grammar for Xcerpt data terms is given in Appendix A.2, full grammars can be found in [Sch04] and [Fur08]. For the semantics of XChange^{EQ}, details on data terms are not important, as we will see in Section 9.2. The set of all data terms is denoted $DataTerms$.[1]

9.1.2 Time

Time is represented by a linearly ordered set $(\mathbb{T}, <)$ of time points (or time instants). Note that this time domain may be dense, in contrast to the time domain of for example CQL which must be discrete (see Chapter 3.3.5). An example of a suitable time domain would be the real numbers under their usual order. This could represent the number of seconds and fractions thereof elapsed since some epoch such as midnight of January 1, 1970.

To support relative temporal events such as `timer:extend`, we also need the possibility to add a duration d to a time point p $(p + d)$ and to subtract a duration from a time point p $(p - d)$. The result in both cases is another time point. Let $\mathbb{D}$ represent durations (such as `1 week 2 days`). Addition and subtraction of durations to or from time points are then functions $+ : \mathbb{T} \times \mathbb{D} \to \mathbb{T}$ and $- : \mathbb{T} \times \mathbb{D} \to \mathbb{T}$.

To support metric temporal constraints such as `within`, we further need the possibility to measure the duration $t_1 - t_2$ between two time points t_1, t_2 and to compare two durations d_1, d_2 $(d_1 < d_2)$. This gives a function $- : \mathbb{T} \times \mathbb{T} \to \mathbb{D}$ and an order relation $<\subseteq \mathbb{D} \times \mathbb{D}$. In context, it will always be clear if $-$ means the subtraction between time points or between a time point and a duration.

A time interval t is in our context always a closed an convex subset of $\mathbb{T}$, i.e., $t = [b, e] = \{p \mid b \leq p \leq e\}$, and can be represented by its endpoints $b \in \mathbb{T}$ and $e \in \mathbb{T}$ $(b \leq e)$. Note that this definition allows degenerated time intervals $t = [p, p]$ that consist only of a single time point $p \in \mathbb{T}$. The set of all time intervals is $\mathbb{TI} = \{[b, e] \mid b \in \mathbb{T}, e \in \mathbb{T}, b \leq e\}$. For convenience we define the following functions and relations on time intervals:

- $begin([b, e]) = b$,
- $end([b, e]) = e$,
- $[b_1, e_1] \sqcup [b_2, e_2] = [\min\{b_1, b_2\}, \max\{e_1, e_2\}]$,
- $[b_1, e_1] \sqsubseteq [b_2, e_2]$ iff $b_2 \leq b_1$ and $e_1 \leq e_2$.

9.1.3 Events

An event happens over a given (closed and convex) time interval and has a representation as data term, which we also call event message. Formally, an event is therefore a tuple of a time interval t and a data term e, written e^t. The set of all events is denoted $Events$; $Events = DataTerm \times \mathbb{TI}$.

9.2 Matching and Constructing Simple Events

To explain how simple event queries are matched against incoming events and how events derived by rules are constructed, we have to explain some concepts of the Web query language Xcerpt, whose query and construct terms are used in XChange^{EQ}. We try too keeps these explanations brief, concentrating on the "interface" to XChange^{EQ} without providing much details about the inner workings of Xcerpt. This also serves to illustrate that the general concepts of XChange^{EQ} and its declarative semantics might easily be applied also to using other languages for querying and constructing simple events.

[1]In [Sch04], $DataTerms$ is written as $\mathcal{T}^d$. We use a more verbose and meaningful notation here.

9.2.1 Substitutions and Substitution Sets

Since query terms and construct terms in rules contain free variables that are bound to values in the application of a rule, we need the concepts of substitution and substitution set.

Let $Vars$ denote the set of all variable names. A **substitution** σ then is a partial mapping from these variable names to the data terms the variables are bound to, i.e., $\sigma : Vars \to DataTerms$. We write substitutions as $\sigma = \{X_1 \mapsto v_1, \dots, X_n \mapsto v_n\}$, meaning that $\sigma(X_i) = v_i$ for $i \in \{1, \dots, n\}$ and $\sigma(Y) = \bot$ for $Y \notin \{X_1, \dots, X_n\}$.

The **application** of a substitution σ to a query term q replaces the occurrences of free variables V in q with their values $\sigma(V)$. The result is denoted $\sigma(q)$. If $\sigma(q)$ is a ground query term, that is, a query term that does not contain variables anymore, we call σ a **grounding substitution**. In the application of σ to q we always assume that σ is defined on all free variables of q.

To accommodate for grouping and aggregation in construct terms, our model theory will mostly work with **substitution sets** Σ, which are just sets of substitutions. For convenience, we define the application of a substitution set Σ to a query term q as $\Sigma(q) = \{\sigma(q) \mid \sigma \in \Sigma\}$. The application of a substitution set to a construct term will be discusses later.

9.2.2 Matching: Simulation

Simple event queries in XChange^{EQ} are single Xcerpt query terms q that are matched against the data term part e of incoming events e^t. This matching of simple event queries is based on simulation between ground terms —that is, terms not containing free variables— as defined for Xcerpt [Sch04].

Simulation is a relation between ground terms denoted $\preceq$. Intuitively, $q \preceq d$ means that the nodes and the structure of the graph that the query term q represents can be found in the graph of the data term d. This simulation relationship of Xcerpt is especially designed for the variations and incompleteness in semi-structured data. We will not reproduce the full definition of simulation here and refer to [Sch04]; for the understanding of the semantics of XChange^{EQ} the intuitive understanding of matching should suffice and we can treat simulation as a "black box."

Simulation naturally extends to a non-ground query term q' by asking whether there is a (grounding) substitution such that the ground query terms $q = \sigma(q')$ obtained by applying the substitution σ to q' simulates with the given data term d. Note that for a given (non-ground) query term q and a given data term d, there are often several substitutions that allow a simulation between the two.

The result of matching a query term q against a data term d thus is the set of all possible substitutions so that q and d simulation under each substitution. An empty substitution set $\sigma \neq \emptyset$ means that q and d do not match. Substitution sets are also the results of complex event queries, which include several query terms as simple event queries.

Example The query term

$q =$ `a {{ desc var X, e [ "f", var Y ] }}`

matches the data term

$d =$ `a [ b {"c"}, "d", e [ "f", "g" ] ]`

with substitution set $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, where

$$\begin{array}{lcll} \sigma_1 & = & \{X \mapsto \texttt{b}\{\texttt{"c"}\}, & Y \mapsto \texttt{"f"}\}, \\ \sigma_2 & = & \{X \mapsto \texttt{"c"}, & Y \mapsto \texttt{"f"}\}, \\ \sigma_3 & = & \{X \mapsto \texttt{"d"}, & Y \mapsto \texttt{"f"}\}. \end{array}$$

9.2.3 Construction: Application of Substitution Sets

Rule heads contain Xcerpt construct terms for constructing new, derived events. This construction uses the set of substitutions obtained from the evaluation of the query in the body to replace variables and grouping constructs with values.

Construction is based on the **application** of a substitution set Σ to a construct term c. The result is denoted $\Sigma(c)$ and is, provided that all substitutions in Σ are grounding, a set of data terms. Again, we will not reproduce the full definition of this application here and refer to [Sch04]; for the understanding of the semantics of XChange$^{\mathrm{EQ}}$ the intuitive understanding of construction should suffice and we can treat application of a substitution set to a construct term as a "black box."

Example The application of the substitution set Σ from the previous section to the construct term

$$q = \texttt{z[var Y, var X]}$$

constructs the three data terms

$$\begin{aligned} d_1 &= \texttt{z["f", b \{"c"\}],} \\ d_2 &= \texttt{z["f", "c"],} \\ d_3 &= \texttt{z["f", "d"].} \end{aligned}$$

Its application to

$$q' = \texttt{z\{var Y, all var X\}}$$

constructs the single data term

$$d' = \texttt{z\{ "f", b \{"c"\}, "c", "d" \}.}$$

9.3 Interpretation and Entailment

We can now define interpretations and entailment, which are the core of the model theory of XChange$^{\mathrm{EQ}}$.

9.3.1 Interpretation

An **interpretation** for a given XChange$^{\mathrm{EQ}}$ query, rule, or program is a 3-tuple $M = (I, \Sigma, \tau)$ where:

1. $I \subseteq Events$ is the set of events e^t that "happen," i.e., are either in the stream of incoming events or derived by some deductive rule.

2. $\Sigma \neq \emptyset$ is a grounding substitution set containing substitutions for the "normal" variables (i.e., data variables, but not event identifiers).

3. $\tau : EventIdentifiers \to Events$ is a substitution for the event identifiers, i.e., a mapping from the names of event identifiers $EventIdentifiers$ to $Events$.

The substitution τ for event identifiers is, compared to model theories of traditional, non-event query languages, unusual. It is needed for evaluating temporal conditions, relative temporal events, etc. Since τ signifies the events that contributed to the answer of some query, we also call it an "event trace."

$I, \Sigma, \tau \models (\texttt{event } i : q)^t$	iff exists $e^{t'} \in I$ with $\tau(i) = e^{t'}$, $t' = t$, and for all $e' \in \Sigma(q)$ we have $e' \preceq e$
$M \models (q_1 \wedge q_2)^t$	iff $M \models q_1^{t_1}$ and $M \models q_2^{t_2}$ and $t = t_1 \sqcup t_2$
$M \models (q_1 \vee q_2)^t$	iff $M \models q_1^t$ or $M \models q_2^t$
$I, \Sigma, \tau \models (Q \texttt{ where } C)^t$	iff $I, \Sigma, \tau \models Q^t$ and $W_{\Sigma,\tau}(C) = \mathit{true}$
$I, \Sigma, \tau \models (\texttt{while } j : \texttt{ not } q)^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $t' = t$, and for all $t'' \sqsubseteq t$ we have $I, \Sigma, \tau \not\models q^{t''}$
$I, \Sigma, \tau \models (\texttt{while } j : \texttt{ collect } q)^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $t' = t$, and exist $n \geq 0$, $\Sigma_1, \ldots \Sigma_n$, $t_1 \sqsubseteq t, \ldots t_n \sqsubseteq t$ with $\Sigma = \bigcup_{i=1..n} \Sigma_i$, and for all $i = 1..n$ we have $I, \Sigma_i, \tau \models q^{t_i}$
$I, \Sigma, \tau \models (c \leftarrow Q)^t$	iff (1) $\Sigma'(c)^t \subseteq I$ for Σ' maximal (w.r.t. $\mathit{FreeVars}(Q)$) and τ' such that $I, \Sigma', \tau' \models Q^t$, or (2) $I, \Sigma', \tau' \not\models Q^t$ for all Σ', τ'

Figure 9.1: Entailment relation defining the model theory for XChange$^{\text{EQ}}$

9.3.2 Entailment

The entailment (or satisfaction) $M \models F^t$ of an XChange$^{\text{EQ}}$ expression F over a time interval t in an interpretation M is defined recursively in Figure 9.1 and 9.2.

Figure 9.1 defines the more salient cases of the model theory. For the sake of brevity, XChange$^{\text{EQ}}$ expression in this figure use binary "and" with symbol $\wedge$ and binary "or" with symbol $\vee$ instead of the multi-ary `and{ ...}` and `or{ ...}`. Also, rules are written as $c \leftarrow Q$ instead of `DETECT` c `ON` Q `END`. In the definitions of the last case, we write $\Sigma'(c)^t$ as obvious shorthand for $\{e^t \mid e \in \Sigma'(c)\}$.

Figure 9.2 defines the cases of the relative temporal events. For the sake of brevity, the prefix "`timer:`" and the keyword "`event`" within the relative timer specification have been skipped.

Our entailment relation uses a fixed interpretation W for all conditions that can occur in the `where`-clause of a query. This includes the temporal relations like `before` as well as conditions on data such as arithmetic comparisons. This fixed interpretation of the temporal conditions is another feature of our model theory that is not common in model theories for traditional, non-event query languages.

W is a function that maps a substitution set Σ, an event trace τ, and an atomic condition C to a boolean value (true or false). We usually write Σ and τ in the index. $W_{\Sigma,\tau}$ extends straightforwardly to boolean formulas of conditions. Figure 9.3 gives the definitions of W for the temporal conditions of XChange$^{\text{EQ}}$ that have been described in Chapter 6.8. The definition of W is deliberately left outside the "core model theory" to make it more modular and demonstrate that it is easy to integrate further conditions or even a separate, external temporal reasoner.

9.4 Models

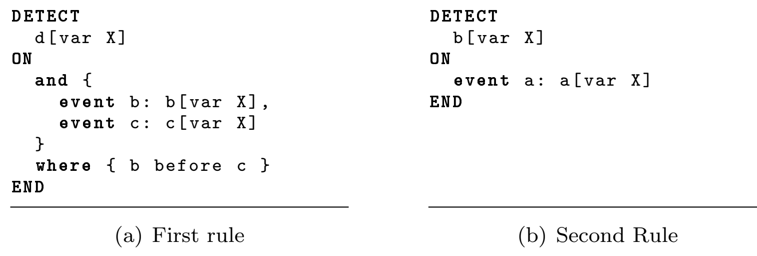
Recall our primary goal in specifying declarative semantics for XChange$^{\text{EQ}}$: given an XChange$^{\text{EQ}}$ program P and an event stream E, we want to find out all events that are derived by the rules in P. This means that we must find an interpretation that contains the event stream E and satisfies all rules of P. Such an interpretation is called a model. However, there are many models (infinitely many, in fact) for a given program P and event stream E. From these models, we will select a single on based on the fixpoint interpretation that is the topic of Chapter 10.

$I, \Sigma, \tau \models (\texttt{event } i : \texttt{extends[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t')$, $end(t) = end(t') + d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{shorten[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t')$, $end(t) = end(t') - d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{extend-begin[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t') - d$, $end(t) = end(t')$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{shorten-begin[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t') + d$, $end(t) = end(t')$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{shift-forward[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t') + d$, $end(t) = end(t') + d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{shift-backward[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t') - d$, $end(t) = end(t') - d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{from-end[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = end(t')$, $end(t) = end(t') + d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{from-end-backward[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = end(t') - d$, $end(t) = end(t')$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{from-start[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t')$, $end(t) = begin(t') + d$
$I, \Sigma, \tau \models (\texttt{event } i : \texttt{from-start-backward[}j, d\texttt{]})^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $\tau(i) = e^t$, $begin(t) = begin(t') - d$, $end(t) = begin(t')$

Figure 9.2: Entailment of relative temporal events in XChange$^{\text{EQ}}$

$W_{\Sigma,\tau}(i$ `before` $j) = \mathbf{true}$	iff $\mathit{end}(\tau(i)) < \mathit{begin}(\tau(j))$
$W_{\Sigma,\tau}(i$ `after` $j) = \mathbf{true}$	iff $\mathit{end}(\tau(j)) < \mathit{begin}(\tau(i))$
$W_{\Sigma,\tau}(i$ `during` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i))$ and $\mathit{end}(\tau(i)) < \mathit{end}(\tau(j))$
$W_{\Sigma,\tau}(i$ `contains` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(i)) < \mathit{begin}(\tau(j))$ and $\mathit{end}(\tau(j)) < \mathit{end}(\tau(i))$
$W_{\Sigma,\tau}(i$ `overlaps` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i)) < \mathit{end}(\tau(j)) < \mathit{end}(\tau(i))$
$W_{\Sigma,\tau}(i$ `overlapped-by` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(i)) < \mathit{begin}(\tau(j)) < \mathit{end}(\tau(i)) < \mathit{end}(\tau(j))$
$W_{\Sigma,\tau}(i$ `meets` $j) = \mathbf{true}$	iff $\mathit{end}(\tau(i)) = \mathit{begin}(\tau(j))$
$W_{\Sigma,\tau}(i$ `met-by` $j) = \mathbf{true}$	iff $\mathit{end}(\tau(j)) = \mathit{begin}(\tau(i))$
$W_{\Sigma,\tau}(i$ `starts` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(i)) = \mathit{begin}(\tau(j))$ and $\mathit{end}(\tau(i)) < \mathit{end}(\tau(j))$
$W_{\Sigma,\tau}(i$ `started-by` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(j)) = \mathit{begin}(\tau(i))$ and $\mathit{end}(\tau(j)) < \mathit{end}(\tau(i))$
$W_{\Sigma,\tau}(i$ `finishes` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i))$ and $\mathit{end}(\tau(i)) = \mathit{end}(\tau(j))$
$W_{\Sigma,\tau}(i$ `finished-by` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(i)) < \mathit{begin}(\tau(j))$ and $\mathit{end}(\tau(j)) = \mathit{end}(\tau(i))$
$W_{\Sigma,\tau}(i$ `equals` $j) = \mathbf{true}$	iff $\mathit{begin}(\tau(i)) = \mathit{begin}(\tau(j))$ and $\mathit{end}(\tau(i)) = \mathit{end}(\tau(j))$
$W_{\Sigma,\tau}(\{i_1,\ldots,i_n\}$ `within` $d) = \mathbf{true}$	iff $E - B \leq d$ with $E := \max\{\mathit{end}(\tau(i_1)), \ldots \mathit{end}(\tau(i_n))\}$ and $B := \min\{\mathit{begin}(\tau(i_1)), \ldots \mathit{begin}(\tau(i_n))\}$.
$W_{\Sigma,\tau}(\{i,j\}$ `apart-by` $d) = \mathbf{true}$	iff $\mathit{end}(\tau(i)) - \mathit{begin}(\tau(j)) \geq d$ or $\mathit{end}(\tau(j)) - \mathit{begin}(\tau(i)) \geq d$.

Figure 9.3: Fixed interpretation for conditions in the `where` clause



```
DETECT
  d[var X]
ON
  and {
    event b: b[var X],
    event c: c[var X]
  }
  where { b before c }
END
```

(a) First rule

```
DETECT
  b[var X]
ON
  event a: a[var X]
END
```

(b) Second Rule

Figure 9.4: Example program P containing two rules

9.4.1 Definition

We now define the notion of a model formally. The definition that follows has a minor caveat: queries in an XChange$^{\text{EQ}}$ program might refer to absolute timer events. These absolute timer events are not part the event stream E and thus need additional treatment. For now, we ignore this and cover it later in Section 9.4.3. In the following, let P be an XChange$^{\text{EQ}}$ program that does not use any absolute timer events.

Given an XChange$^{\text{EQ}}$ program P and a stream of incoming events E, we call an interpretation $M = (I, \Sigma, \tau)$ a **model** of P under E if

- M satisfies all rules $r = (c \leftarrow Q) \in P$ for all time intervals t, i.e., $M \models r^t$ for all $t \in \mathbb{TI}$ and all $r = (c \leftarrow Q) \in P$, and
- M contains the stream of incoming events, i.e., $E \subseteq I$.

On close inspection of the entailment relation, we can see that Σ and τ are actually irrelevant to whether a given interpretation M is a model or not; it depends only on I: M must satisfy all rules and the rule application in the entailment relation (last case in Figure 9.1) does not refer to Σ and τ in its definition (only to the existence or non-existence of some Σ' and τ'). We therefore can identify the notion of a model with just the I-part of the interpretation, $M = I$.

9.4.2 Example

Consider the XChange$^{\text{EQ}}$ program P in Figure 9.4 and the event stream

$$E = \{\texttt{a["z"]}^{[1,2]},\ \texttt{a["y"]}^{[1,3]},\ \texttt{c["z"]}^{[3,5]},\ \texttt{a["z"]}^{[6,9]}\}.$$

The interpretation

$$\begin{aligned} M_1 = \{\ & \texttt{a["z"]}^{[1,2]},\ \texttt{b["z"]}^{[1,2]},\ \texttt{a["y"]}^{[1,3]},\ \texttt{b["y"]}^{[1,3]},\ \texttt{c["z"]}^{[3,5]},\ \texttt{d["z"]}^{[1,5]}, \\ & \texttt{a["z"]}^{[6,9]},\ \texttt{b["z"]}^{[6,9]}\} \end{aligned}$$

is a model for P under E: by applying the recursive definition of $\models$ from Section 9.3 we can check that $M_1 \models r^t$ for all $t \in \mathbb{TI}$, $r \in P$, and we also have $E \subseteq M_1$.

The interpretation

$$\begin{aligned} M_2 = \{\ & \texttt{a["z"]}^{[1,2]},\ \texttt{b["z"]}^{[1,2]},\ \texttt{a["y"]}^{[1,3]},\ \texttt{b["y"]}^{[1,3]},\ \texttt{c["z"]}^{[3,5]},\ \texttt{d["z"]}^{[1,5]}, \\ & \texttt{d["y"]}^{[1,5]},\ \texttt{a["z"]}^{[6,9]},\ \texttt{b["z"]}^{[6,9]},\ \texttt{e["y"]}^{[7,9]}\} \end{aligned}$$

where we "added" the two events $\texttt{d["y"]}^{[1,5]}$ and $\texttt{e["y"]}^{[7,9]}$ in comparison to M_1 is also a model of P under E. Clearly, however, M_2 is not the model we intend for our program to have, because the two "additional" events are "unjustified." More precisely, these two events are neither in the event stream E nor derived by a rule of P. M_1 is the intended model, because all events in it are justified.

To unambiguously settle on a single, intended model, we will use the fixpoint theory that will be defined in the next chapter. The fixpoint theory builds upon the model theory defined in this chapter. Note that the problem of specifying the intended model out of the many possible models is common part of the model-theoretic approach. It is not specific to event query languages and exists also with all model theories of traditional, non-event query languages.

9.4.3 Incorporating Absolute Timer Events

We have yet to explain how to incorporate absolute timer events into the model theory. One possibility would of course be to extend our definition of the entailment relation to have one or more cases for the queries to absolute timer events. This approach has already been used for the relative timer events. However the right hand side of this definition would be rather involved and the approach would not be very modular in case we want to embed other calendric systems than the built-in `timer:datetime` of XChange$^{\text{EQ}}$.

Instead we use a more modular approach. Absolute timers are modeled as a (usually infinite) set C of events, that is, of data terms that are associated with an occurrence interval. We call C a **calendar specification**.

If our time domain $\mathbb{T}$ relates to real time as seconds (or fractions thereof) elapsed since midnight of January 1, 1970 (UTC) and the local time zone is Central European Time (CET), then C would for example contain the following events with an occurrence time $t = [0, 0]$ (and more events for the other time zones):

```
timer:datetime {
  year {1970},
  month {"January"},
  day-of-month {1},
  hour-of-day {0},
  minute {0},
  second {0},
  millisecond {0},

  week-of-year {1},
  week-of-month {1},
  day-of-year {1},
  day-of-week {"Friday"},
  day-of-week-in-month {1},
  am-pm {"am"},
  hour {0},
  zone-offset {0},
  dst-offset {0},

  date {"1970-01-01"},
  time {"00:00Z"},
}
```

```
timer:datetime {
  year {1970},
  month {"January"},
  day-of-month {1},
  hour-of-day {1},
  minute {0},
  second {0},
  millisecond {0},

  week-of-year {1},
  week-of-month {1},
  day-of-year {1},
  day-of-week {"Friday"},
  day-of-week-in-month {1},
  am-pm {"am"},
  hour {1},
  zone-offset {1},
  dst-offset {0},

  date {"1970-01-01"},
  time {"01:00"},
}
```

```
timer:datetime {
  year {1970},
  month {"January"},
  day-of-month {1},
  hour-of-day {1},
  minute {0},
  second {0},
  millisecond {0},

  week-of-year {1},
  week-of-month {1},
  day-of-year {1},
  day-of-week {"Friday"},
  day-of-week-in-month {1},
  am-pm {"am"},
  hour {1},
  zone-offset {1},
  dst-offset {0},

  date {"1970-01-01"},
  time {"01:00+01:00"},
}
```

With this approach, treatment of absolute timer events is now reduced to entailment of simple event queries (first case in Figure 9.1). We only have to adapt or notion of model as follows:

Let C be a calendar specification as described above. Given a an XChange$^{\text{EQ}}$ program P and a stream of incoming events E, we call an interpretation $M = (I, \Sigma, \tau)$ a **model** of P under E if

- M satisfies all rules $r = (c \leftarrow Q) \in P$ for all time intervals t, i.e., $M \models r^t$ for all $t \in \mathbb{TI}$ and all $r = (c \leftarrow Q) \in P$, and
- M contains the stream of incoming events and the calendar specification, i.e., $C \cup E \subseteq I$.

Chapter 10

Fixpoint Theory

In the previous chapter we have defined a model theory for XChange^{EQ} programs. As we have seen in Chapter 9.4.2, however, there are many models for a given program. A common and convenient way to obtain a unique model is to define a fixpoint theory. The intended model is the (least) fixpoint of the immediate consequence operator, which derives new events from known events (based on the model theory).

Non-monotonic features such as negation and aggregation introduce well-known issues when they are combined with recursion of rules. In particular, there might be not be a fixpoint or several. To ensure that a single fixpoint exists, we restrict XChange^{EQ} programs to be stratifiable. This is a common approach from logic programming introduced first in [ABW88].

In addition to giving unambiguous semantics to stratifiable XChange^{EQ} programs, the fixpoint theory also describes an abstract, simple, forward-chaining evaluation method, which can easily be extended to work incrementally as is required for event queries (see Chapter 12).

10.1 Stratification: Limits on Recursion

Consider the event stream $E = \{\texttt{s[]}^{[1,2]}\}$ and following program P, which has already been discussed in Chapter 6.10:

```
DETECT
  p [ ]
ON
  and {
    event s: s [ ],
    while s: not q [ ]
  }
END
```

```
DETECT
  q [ ]
ON
  and {
    event s: s [ ] ,
    while s: not p [ ]
  }
END
```

Both $M_1 = \{\texttt{s[]}^{[1,2]}, \texttt{p[]}^{[1,2]}\}$ and $[M_2 = \{\texttt{s[]}^{[1,2]}, \texttt{q[]}^{[1,2]}\}$ are models of P under E. Because the two models are symmetric, there is also no clear criterion to select one or the other as intended model of this XChange^{EQ} program. This is a common and inherent difficulty when rules and negation are combined. (It is in fact just an adaption of the standard non-event example $p \leftarrow \neg q, q \leftarrow \neg p$ from logic programming and deductive databases.) A simple and established solution is to avoid such situations by requiring programs to be stratifiable.

Stratification restricts the use of recursion in rules by ordering the rules of a program P into so-called strata (sets P_i of rules with $P = P_1 \uplus \cdots \uplus P_n$) such that a rule in a given stratum can only depend on (i.e., access results from) rules in lower strata (or the same stratum, in some cases).

Three types of stratification are required:

1. Negation stratification: Events that are negated in the query of a rule may only be constructed by rules in lower strata. Events that occur positively may only be constructed by rules in lower strata or the same stratum.

2. Grouping stratification: Rules using grouping constructs like `all` in the construction may only query for events constructed in lower strata.

3. Temporal stratification: If a rule queries a relative temporal event like `timer:extends[event i, 1min]` then the anchoring event (here: `i`) may only be constructed in lower strata.

While negation and grouping stratification are fairly standard, temporal stratification is a requirement specific to complex event query programs like those expressible in XChange^{EQ}. We are not aware of former consideration of the notion of temporal stratification.

To define stratification, we first need the notion of dependency.

10.1.1 Rule dependencies

We say that some rule r depends on another rule r' if r (potentially) queries events that have been constructed by r'. We distinguish different kinds of dependency:

- $r = c \leftarrow Q \in P$ *depends temporally* on $r' = c' \leftarrow Q' \in P$ if there exists a query term q in Q with an event identifier j attached to it such that j is used elsewhere in Q to define a relative temporal event and $q \preceq c'$.[1]

- $r = c \leftarrow Q \in P$ *grouping depends* on $r' = c' \leftarrow Q' \in P$ if c contains grouping constructs (such as `all`) and there exists a query term q in Q such that $q \preceq c'$.

- $r = c \leftarrow Q \in P$ *depends negatively* on $r' = c' \leftarrow Q' \in P$ if there exists a query term q that occurs negated (i.e., within a `not`) in Q such that $q \preceq c'$.

- $r = c \leftarrow Q \in P$ *depends positively* on $r' = c' \leftarrow Q' \in P$ if r does not depend on r' by the previous dependencies and there exists a query term q in Q such that $q \preceq c'$.

Note that positive, negative, and grouping dependencies are analogously defined for the (non-event) query language Xcerpt [Sch04] that underlies XChange^{EQ}.

10.1.2 Stratified Programs

A **stratification** $P = P_1 \uplus \cdots \uplus P_n$ for an XChange^{EQ} program P is a partitioning of the rules of P such that for each pair of rules $r = c \leftarrow Q \in P_i$, $r' = c' \leftarrow Q' \in P_j$:

- if r depends temporally on r', then $i > j$,

- if r grouping depends on r', then $i > j$,

- if r depends negatively on r', then $i > j$

- if r depends positively on r', then $i \geq j$.

An XChange^{EQ} program is called **stratifiable**, if there exists a stratification for it. By building a dependency graph [ABW88] as outlined in Chapter 11.3.4, we can check if a given XChange^{EQ} is in fact stratifiable and obtain a stratification for it. The dependency graph is sometimes also called a precedence graph [AHV95].

---

[1] In slight abuse of notation, we write $q \preceq c'$ for $\exists \Sigma \exists e \in \Sigma(q) \exists e' \in \Sigma(c). e \preceq e'$.

10.1.3 Hierarchical Programs

To obtain more efficient evaluation algorithms, it might be desirable to restrict the use of recursion in XChange^{EQ} programs even further. Hierarchical programs do not allow any recursion cycles [Llo93].

An XChange^{EQ} program is called **hierarchical**, if there exists a partitioning $P = P_1 \uplus \cdots \uplus P_n$ of the rules of P such that for each pair of rules $r = c \leftarrow Q \in P_i$, $r' = c' \leftarrow Q' \in P_j$:

- if r depends temporally on r', then $i > j$,
- if r grouping depends on r', then $i > j$,
- if r depends negatively on r', then $i > j$
- if r depends positively on r', then $i > j$.

Note that the difference between stratifiable and hierarchical programs is in the last line of the definition: stratifiable programs allow positive recursion cycles within a stratum, hierarchical programs exclude this. Obviously, every hierarchical program is stratifiable.

For our declarative semantics, the restriction to hierarchical programs brings no additional benefit, so we treat the more general case of stratifiable XChange^{EQ} programs. The operational semantics for XChange^{EQ} will focus on hierarchical programs, but an extension to stratifiable programs is straightforward (see Chapter 18.1.1).

10.1.4 Remarks

The restriction to stratifiable programs is necessary for the fixpoint semantics given in the next section. Note that this is not a very severe restriction in the domain of event queries.

It would also be conceivable to (partially) lifted the restriction to stratifiable programs at the cost of a more involved semantics and evaluation. Approaches to this have been explore in depth in research related to logic programming and deductive databases.

Extending the semantics of XChange^{EQ} beyond stratified programs is possible with the established approaches from logic programming, but outside the scope of this work. See also Chapter 18.

10.2 Immediate Consequence Operator

The basic idea for obtaining the fixpoint interpretation of a stratified XChange^{EQ} program is to apply the rules stratum by stratum: first apply the rules in the lowest stratum to the incoming event stream, then apply the rules in the next higher stratum to the result, and so on until the highest stratum.

This requires the definition of the **immediate consequence operator** T_P for an XChange^{EQ} program. It is defined as:

$$\begin{aligned} T_P(I) = I \cup \{e^t \mid \quad & \text{there exist a rule } c \leftarrow Q \in P, \text{ a maximal substitution set } \Sigma, \\ & \text{and a substitution } \tau \text{ such that } I, \Sigma, \tau \models Q^t \text{ and } e \in \Sigma(c)\} \end{aligned}$$

The repeated application of T_P until a fixpoint is reached is denoted T_P^ω. A fixpoint here means an interpretation I such that $T_P(I) = I$. As we will see in Chapter 11.3, T_P^ω is also a *least* fixpoint.

Because we are interested in fixpoints of the immediate consequence operator, it is sometimes also called fixpoint operator.

10.3 Fixpoint Interpretation

Let $\overline{P_i} = \bigcup_{j \leq j} P_j$ denote the set of all rules in strata P_i and lower. The **fixpoint interpretation** $M_{P,E}$ of an XChange$^{\text{EQ}}$ program P with stratification $P = P_1 \uplus \cdots \uplus P_n$ under event stream E is defined by computing fixpoints stratum by stratum:

$$\begin{array}{rcrcl} & & M_0 & = & E = T^{\omega}_{\emptyset}(E), \\ & & M_1 & = & T^{\omega}_{\overline{P_1}}(M_0), \\ & & \ldots, & & \\ M_{P,E} & = & M_n & = & T^{\omega}_{\overline{P_n}}(M_{n-1}). \end{array}$$

The fixpoint interpretation $M_{P,E}$ is also called the **intended model** of P under E and specifies the declarative semantics. The first theorem in the next chapter will show that these semantics are, in fact, well-defined and unambiguous and thus justifies our definition. As we will see, a similar theorem must usually proven also for fixpoint semantics of non-event query languages.

Fixpoint semantics straight-forwardly outline a forward-chaining evaluation of queries, which is the evaluation method of choice for event query programs (see Chapter 12). The main difference between fixpoint semantics and an actual evaluation method is that the actual evaluation should work incrementally, i.e., operate only on the part of the event stream received so far. The second theorem in the next chapter will show that this is, in fact, possible. Note that this is not self-evident: a regular, non-event query language that supports non-monotonic features such as negation (e.g., Xcerpt) could not be evaluated in a manner where only part of the base facts (which correspond to the event stream) are known.

Chapter 11

Theorems

We now give two theorems about the declarative semantics of XChange$^{\mathrm{EQ}}$ that have been specified in Chapters 9 and 10. The first theorem shows that the semantics provided by the fixpoint interpretation are well-defined and unambiguous. The second theorem shows that the semantics are suited for infinite event streams and justify a streaming evaluation.

We first state and explain the two theorems (Sections 11.1 and 11.2) and then give their proofs (Sections 11.3 and 11.4).

11.1 Well-Defined and Unambiguous Semantics

The following theorem states that the semantics provided by the fixpoint interpretation defined in Chapter 10.3 are well-defined and unambiguous. It also shows that they correspond to our natural intuition in that all events it contains are justified by either being in the incoming event stream or being derived by a rule.

Theorem 1 For a stratifiable program P and an event stream E, $M_{P,E}$ is a minimal model of P under E. Further, $M_{P,E}$ is independent of the stratification of P.

"Minimal" in the theorem entails that all events in the model are either in the stream of incoming events or have been derived by rules, i.e., no events have been added without justification.

Note that this theorem is not specific to event query languages. Similar theorems must be shown also for non-event query languages. In fact the proof given in Section 11.3 is an adaption of a standard proof from [Llo93] to the query and construct terms of Xcerpt.

While the model and fixpoint theory of XChange$^{\mathrm{EQ}}$ bears some resemblance to that of the underlying query language Xcerpt, however, such a theorem has not been proven for Xcerpt so far (cf. Chapter 7 of [Sch04]). The proof of the theorem for XChange$^{\mathrm{EQ}}$ given in Section 11.3 also applies to Xcerpt if the temporal aspects of XChange$^{\mathrm{EQ}}$'s semantics are ignored. It thus also fills in this minor gap in the semantics of Xcerpt.

11.2 Suitability for Event Streams

More interestingly in the context of event queries, we must and can show that the model theory and fixpoint semantics are sensible on infinite event streams. The next theorem justifies a streaming evaluation, where answers to complex event queries are generated "online" and we never have to wait for the stream to end. This is especially important since event streams can conceptually be infinite and thus not end at all.

In particular it ensures an event e^t can be detected at the time point $end(t)$ since no knowledge about any events in the future of $end(t)$ is required. Ensuring that evaluation methods are not

expected to "crystal gaze" is of course an important requirement and one example where we can use the declarative semantics to prove interesting statements about a (complex) event query language.

Theorem 2 Let $E \mid t$ denote the restriction of an event stream E to a time interval t, i.e., $E \mid t = \{e^{t'} \in E \mid t' \sqsubseteq t\}$. Similarly, let $M \mid t$ denote the restriction of an interpretation M to t. Then the result of applying the fixpoint procedure to $E \mid t$ is the same as applying it to E for the time interval t, i.e., $M_{P,E|t} \mid t = M_{P,E} \mid t$.

Simply stated, the theorem says that in order to evaluate a program over a time interval t, we do not have to consider any events happening outside of t.

11.3 Proof of Theorem 1

We now turn to the proof of the first theorem.

11.3.1 Minimal Model

We want to prove that the fixpoint interpretation $M_{P,E}$ of a stratified XChange$^{\text{EQ}}$ program P under an event stream E is a minimal model, i.e., there is no model M' of P under E with $M' \subsetneq M_{P,E}$. An analogous statement for stratified logic programs is well-established [ABW88]. For our proof in the world of XChange$^{\text{EQ}}$, we adapt the proof from [Llo93].[1] The most important hurdle in transferring this proof is that the stratification of XChange$^{\text{EQ}}$ programs is defined differently: in XChange$^{\text{EQ}}$ strata consist of rules, while in logic programming strata consist of predicate symbols.

Before starting with our proof, we repeat a well-known result on fixpoints due to Tarski (as presented in [Llo93]): Let L be a complete lattice with order $\leq$ and $T : L \to L$ monotonic. Then T has a least fixpoint, $lfp(T)$, that is the greatest lower bound (glb) of the set of all fixpoints of T as well as of the set of all prefixed points: $lfp(T) = glb\{x \mid T(x) = x\} = glb\{x \mid T(x) \leq x\}$.

Let $P = P_1 \uplus \cdots \uplus P_n$ be a stratified XChange$^{\text{EQ}}$ program and E an event stream. We show by induction on the number n of strata that $M_{P,E} = M_n$ is a minimal fixpoint of T_P with $E \subseteq M_{P,E}$. For the induction base $n = 0$ this is obvious since then $P = \emptyset$ and thus T_P is simply the identity transformation.

For the induction step $n - 1 \to n$, we will make use of Tarski's theorem from above. Of course we cannot apply the theorem to T_P directly since T_P is, in general, not monotonous. However we can apply it to a certain restriction. Let Λ be the following complete lattice:

$$\Lambda = \{M_{n-1} \cup S \mid S \subset \bigcup_{r \in P} gi(r)\}$$

Here $gi(r)$ denotes the set of all events a rule $r = c \leftarrow Q$ can query or construct. Speaking in the language of logic programming, the "*g*round *i*nstances" of the terms occurring in the rule[2]. Formally:

$$\begin{aligned} gi(c \leftarrow Q) \quad = \quad & \{e^t \mid t \in \mathbb{TI} \text{ and } \exists \Sigma . e \in \Sigma(c)\} \cup \\ & \{e^t \mid t \in \mathbb{TI} \text{ and } \exists \text{query term } q \text{ in } Q \ \exists \sigma . e \preceq \sigma(q)\} \end{aligned}$$

Lemma 1 The restriction $T_P \mid_\Lambda$ of T_P to the complete lattice Λ is well-defined (i.e., application of T_P to an element of Λ yields an element that is again in Λ) and monotonous (i.e., $I \subseteq J$ implies

[1] As we will see, the proof in [Llo93] uses the existence of least fixpoints for monotonic operators on complete lattices, a well-known result established by Knaster and Tarski. The earlier proof in [ABW88] works without this result. We base our proof on [Llo93] because it is shorter and seems more intuitive and easier to understand than [ABW88].

[2] To avoid any confusion: note that we talk about the ground instances of the *terms* occurring in the rule, not about ground instances of the rule itself.

$T_P \mid_\Lambda (I) \subseteq T_P \mid_\Lambda (J)$). To avoid distraction from the main proof, we give the proof for this lemma later.

With Tarski's theorem from above, this lemma gives us that $T_P \mid_\Lambda$ has a least fixpoint M:

$$\begin{aligned} M = \quad lfp(T_P \mid_\Lambda) &= glb\{I \in \Lambda \mid T_P(I) = I\} \\ &= glb\{I \in \Lambda \mid T_P(I) \subseteq I\} \end{aligned}$$

Note that $M = M_{P,E}$ simply by definition of $M_{P,E} = T_P^\omega(M_{n-1})$. Further, M is a model for P under E due to the following Lemma.

Lemma 2 An interpretation I is a model for an XChange$^{\mathrm{EQ}}$ program P ("$I \models P$") if and only if $T_P(I) \subseteq I$. Again, we delay the proof for this lemma.

We now show that M is minimal, i.e., if some $M' \subseteq M$ is a model of P and $E \subseteq M'$ then $M' = M$. By induction hypothesis, we have that $M_{n-1} \subseteq M'$ (since M_{n-1} is a minimal model for $P_1 \uplus \cdots \uplus P_{n-1}$). By definition of Λ this gives us that $M' \in \Lambda$. Lemma 2 and M being also the least pre-fixed point of $T_P \mid_\Lambda$ yield $M' \subseteq M$.

11.3.2 Proof of Lemma 1

To show that $T \mid_\Lambda$ is well-defined, let $I \in \Lambda$. By definition of Λ, $I = M_{k-1} \uplus S_I$ for some $S_I \subseteq \bigcup_{r \in P} git(r)$. Now $T_P(I) = I \cup \{e^t \mid e^t$ generated by some rule $r \in P\}$ and the right side of the union is a subset of $\bigcup_{r \in P} git(r)$. This gives us that $T_P(I) = M_{k-1} \uplus S_T$ for some $S_T \subseteq \bigcup_{r \in P} git(r)$ (S_T is the union of $\{e^t \mid \dots\}$ and S_I) and thus $T_P(I) \in \Lambda$.

For $T_P \mid_\Lambda$ monotonic, let $I \in \Lambda$, $J \in \Lambda$, $I \subseteq J$ and $e^t \in T_P(I)$. What we want to show is that $e^t \in T_P(J)$.

If $e^t \in I$ we immediately have $e^t \in T_J(P)$ by $I \subseteq J$ and the definition of T_P. Otherwise there is a rule $r = c \leftarrow Q \in P$ and a maximal Σ and a τ with $I, \Sigma, \tau \models Q$ and $e^t \in \Sigma(c)$. If $r \in \overline{P_{n-1}}$, then $e^t \in M_{n-1}$ and thus also $e^t \in T_P(J)$ since $M_{n-1} \subseteq T_P(J)$ (remember that $T_P \mid \Lambda$ is well-defined). It remains to consider the case where $r \in P_n$, where we have to show that $e^t \in \Sigma(c)$.

We distinguish whether c is free of grouping constructs or not. In the former case it suffices to show that $J, \Sigma, \tau \models Q^t$. In the latter case we have to show additionally that Σ is maximal.

Case 1: c free of grouping constructs. By induction on Q we show that $J, \Sigma, \tau \models Q^t$. Besides the induction hypothesis (IH) and the definition of the model theory (Def$_\models$) from Figure 9.1, we have available that $I, \Sigma, \tau \models Q^t$ with Σ being maximal $(*)$ and that $I \subseteq J$ $(**)$.

Case 1.1: $Q =$ (event $i : q$). By (Def$_\models$) and $(*)$, we have $e^t \in I$ and by $(**)$ we get the $e^t \in J$.

Case 1.2: $Q =$ (event i : extends$[j, d]$). Trivial, since (Def$_\models$) makes no reference to the interpretation I (J, respectively). As in Figure 9.1, we skip the other temporal events since they are analogous.

Case 1.3: $Q = (q_1 \wedge q_2)$. (Def$_\models$) and $(*)$ gives us $I, \Sigma, \tau \models q_1^{t_1}$ and $I, \Sigma, \tau \models q_2^{t_2}$ with $t = t_1 \sqcup t_2$. Applying (IH) we get $J, \Sigma, \tau \models q_1^{t_1}$ and $J, \Sigma, \tau \models q_2^{t_2}$ and can apply (Def$_\models$).

Case 1.4: $Q = (q_1 \vee q_2)$. Obvious application of (IH), see case 1.3.

Case 1.5: $Q = (Q'$ where $C)$. Obvious application of (IH).

Case 1.6: $Q =$ (while j : not q). $(*)$ gives us an e^t with $\tau(j) = e^t$ and $I, \Sigma, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. We have to show that also $J, \Sigma, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. Now, if there were a t'' such that $J, \Sigma, \tau \models q^{t''}$, then already $M_{n-1}, \Sigma, \tau \models q^{t''}$ due to the (negation) stratification. This however would imply $I, \Sigma, \tau \models q^{t''}$ in contradiction to our assumptions.

Case 1.7: $Q =$ (while j : collect q). Again a simple application of (IH).

Case 2: c contains grouping constructs. As in case 1 we get that $J, \Sigma, \tau \models Q^t$. It remains to show by induction on Q that Σ is in fact maximal, i.e., $J, \Sigma \cup \{\sigma\}, \tau \models Q^t$ with some $\sigma \notin \Sigma$ $(***)$ leads to a contradiction with the maximality in $(*)$.

Case 2.1: $Q =$ (event $i : q$). Suppose by (Def$_\models$) and the assumption $(***)$ that there is an $e^t \in J$ with $\sigma(q) \preceq e$. Due to stratification, $e^t \in M_{n-1}$ and thus $I, \Sigma \cup \{\sigma\}, \tau \models Q^t$ in contradiction to $(*)$.

Case 2.2: $Q =$ (event i : extends$[j, d]$). Trivial since $I, \Sigma, \tau \models Q^t$ for any Σ, i.e., also for $\Sigma \cup \{\sigma\}$.

Case 2.3: $Q = (q_1 \wedge q_2)$. (Def$_\models$) and $(***)$ give $J, \Sigma \cup \{\sigma\}, \tau \models q_1^{t_1}$ and $J, \Sigma \cup \{\sigma\}, \tau \models q_2^{t_2}$ and $t = t_1 \sqcup t_2$. Application of (IH) leads to the contradiction.

Case 2.4: $Q = (q_1 \vee q_2)$. Obvious application of (IH), see case 1.3.

Case 2.5: $Q = (Q'$ where $C)$. Obvious application of (IH).

Case 2.6: $Q =$ (while j : not q). (Def$_\models$) and $(***)$ give an e^t with $\tau(j) = e^t$ such that $J, \Sigma \cup \{\sigma\}, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. The maximality in $(*)$ however gives $I, \Sigma \cup \{\sigma\}, \tau \models q^{t''}$. By (IH) then the contradiction $J, \Sigma, \tau \models q^{t''}$.

Case 2.7: $Q =$ (while j : collect q). By (Def$_\models$) and $(***)$ there must exist an e^t with $\tau(j) = e^t$ such that there are Σ_i and $t_i \sqsubset t$ with $J, \Sigma_i \cup \{\sigma\}, \tau \models q^{t_i}$. Application of (IH) now gives the contradiction.

11.3.3 Proof of Lemma 2

We want to show that $M \models P$ if and only if $T_P(M) \subseteq M$. From right to left, suppose $T_P(M) \subseteq M$, but $M \not\models P$, i.e., there is s a rule $r = c \leftarrow Q \in P$ with $M \not\models r$. Accordingly we must have a $t \in \mathbb{T}$, a τ, and a maximal Σ with $M, \Sigma, \tau \models Q^t$ but $\Sigma(c)^t \nsubseteq M$. I.e., there must be an $e \in \Sigma(c)$ such that $e \notin M$, which however is in contradiction to $e \in T_P(M) \subseteq M$.

From left to right, let $M \models P$ and $e^t \in T_P(M)$ and suppose $e^t \notin M$. Then e^t must have been generated by a rule $r = c \leftarrow Q \in P$. Accordingly we must have a τ and a maximal Σ such that $M, \Sigma, \tau \models Q^t$ and $e \in \Sigma(c)$. This however would mean that $e^t \in M$ since $M \models r$, which gives us the contradiction.

11.3.4 Independence from Stratification

To prove that $M_{P,E}$ is said to be independent of the given stratification of P, we show that any two possible stratifications of P are equivalent, i.e., the fixpoint procedure yields the same model. In the world of stratified logic programs, this is again a well-established result. In fact, the roof of this statement for datalog$^\neg$ found in [AHV95] (Theorem 15.2.10) transfers directly to XChange$^{\text{EQ}}$. Only two things need to be adapted to deal with XChange$^{\text{EQ}}$'s notion of stratification, which is defined over rules not predicate symbols: the notion of the precedence graph (sometimes this is also called a dependency graph) and a lemma that enables us to argue that if two strata that are independent of each other then they can be permuted in the fixpoint procedure. We do not repeat the proof from [AHV95] here, but give only the two adaptations just mentioned.

The **precedence graph** G_P for an XChange$^{\text{EQ}}$ program P is a directed graph with edges labeled either "+" (called positive edges) or "−" (called negative edges). The vertices of the graph are the rules of P. There is a positive edge from r to r' if r depends positively on r'. There is a negative edge from r to r' if r' depends on r' in any other way (negatively, temporally, or by grouping).[3]

We have to show the following **lemma** as a replacement for lemma 15.2.9 in [AHV95] (note that this is the only point where the proof of [AHV95] is specific to datalog$^\neg$): If P is a semi-positive XChange$^{\text{EQ}}$ program, i.e., it only contains positive dependencies, and $P = P_1 \uplus P_2$ is a stratification of P, then the fixpoint procedure yields the same model for P and for $P_1 \uplus P_2$: $T_P^\omega(E) = T_{\overline{P_2}}^\omega(T_{\overline{P_1}}^\omega(E))$ for all event streams E.

Proof. Observe that $T_P = T_{\overline{P_2}}$ is monotonous (just like in Lemma 1 from above). With the inclusion $E \subseteq T_{P_1}^\omega(E) = T_{\overline{P_1}}^\omega(E)$ this yields $T_P^\omega(E) \subseteq T_{\overline{P_2}}^\omega(T_{\overline{P_1}}^\omega(E))$. On the other hand, the inclusion $T_{\overline{P_1}}^\omega(E) = T_{P_1}^\omega(E) \subseteq T_P^\omega(E)$ gives $T_{\overline{P_2}}^\omega(T_{\overline{P_1}}^\omega(E)) \subseteq T_{\overline{P_2}}^\omega(T_P^\omega(E)) = T_P^\omega(T_P^\omega(E)) = T_P^\omega(E)$.

[3]Admittedly, we are a bit abusive of notation here, using "−" to label not only negative dependencies but also grouping and temporal dependencies. However, it is not necessary to distinguish negative, grouping, and temporal dependencies here, since also in the definition of a stratification the are treated the same (requiring the dependent rule to be in a strictly higher stratum).

11.4 Proof of Theorem 2

We have to show that $M_{P,E|u} \mid u = M_{P,E} \mid u$ for an arbitrary time interval u. For this, we first make the following observation.

Lemma 3 Let t and u be time intervals with $t \sqsubseteq u$ and let Q be a query. We then have:

$$I \mid u, \Sigma, \tau \models Q^t \quad \text{iff} \quad I, \Sigma, \tau \models Q^t$$

The proof for this is by a trivial induction on Q. With the definition of the immediate consequence operator T_P, the above observation gives us that

$$T_P(I \mid u) \mid u = T_P(I) \mid u$$

for all time intervals u and all programs P.

Further, the definition of T_P says that all events e^t constructed by a rule r "inherit" their occurrence time t from the rule's query. Thus it holds that $(T_P \mid u)^\omega = (T_P^\omega) \mid u$ and we get $M_{P,E|u} \mid u = M_{P,E} \mid u$.

Part IV

Incremental Evaluation of Complex Event Queries

Chapter 12

Operational Semantics: Requirements and Overview

While the declarative semantics developed in the preceding chapters give XChange^{EQ} programs a clear and formal meaning, they offer little help for the actual evaluation of XChange^{EQ} complex event query programs. The following chapters develop operational semantics that describe how XChange^{EQ} programs can be evaluated efficiently. The operational semantics provide an abstract description of an implementation of an XChange^{EQ} evaluation engine and are a basis for query optimization.

In this chapter, we explain the basic concepts involved in evaluating event queries and the motivation and ideas behind the operational semantics of XChange^{EQ}. Chapters 13, 14, and 15 will then fully develop the operational semantics. Chapter 16 describes the prototype implementation of the XChange^{EQ} engine that is based on these operational semantics. The results of these chapters have been presented with a strong focus on temporal relevance in [BE08a]. Earlier work towards operational semantics for XChange^{EQ} can be found in [BE07b] and [BE07c].

12.1 Basics of Event Query Evaluation

Event query evaluation means evaluating standing (complex) event queries against a stream of incoming events. For each incoming event, we have to check if this event together with some of the events received previously leads to new answers for the standing event query. Records of these previous events must be maintained in the event query evaluation engine in a data structure commonly called event history. In addition to checking for new answers, therefore, we also have to update this event history with the current event in order to prepare it for future incoming events. Separately or as part of updating the event history, events in the history that have become irrelevant over time must be removed to free up their memory.

Event query evaluation has much in common with traditional query evaluation in databases or on Web documents, as this work tries to emphasize. There are however also important differences:

- Event queries are standing queries that are evaluated against event data that is coming in as a streams. In a database, data is "standing" (or maybe rather lying around) and queries are coming in to be evaluated.
- Accordingly, event query evaluation is a step-wise process over time, where each step is initiated by an incoming event. In contrast, traditional query evaluation is a one-time process initiated by an "incoming" query.
- Event streams conceptually stretch out infinitely into the future and at each point in time at most the history of the stream up to that point is known. Database data is finite and all data is known.

- Event query evaluation has to actively maintain a history of events that have been received so far from the event stream and perform garbage collection in this history to free up memory from irrelevant events. Database query evaluation does not have a need for such histories or for garbage collection.

- Timing and order of answers to event queries, i.e., of detected complex events, play an important role because actions that might be triggered as response to complex events are generally sensitive to execution time and order. Answers to a database queries are conceptually delivered all at once and there is no notion of order or timing in the data.

- Efficient query evaluation and query optimization have some different assumptions and require some different techniques. In the compilation of event queries no or only little information about the data distribution characteristics of the event stream are available and an important optimization technique is to exploit similarities between different queries (multi-query optimization). The evaluation is usually done in main memory, where the difference between sequential and random access is fairly small. In the compilation of database queries a fair amount of information about the data distribution characteristics is available and multi-query optimization is not that relevant. The evaluation retrieves data from a hard disc and puts emphasis on having few and sequential page accesses.

We now detail the most important aspects of event query evaluation further.

12.1.1 Step-Wise Evaluation over Time

Evaluating an event query or an event query program (which contains several rules and thus several event queries) is a step-wise procedure. Evaluation is done over time so that each evaluation step is associated with a time point *now* at which it is executed.

An evaluation step at time *now* is initiated by incoming events. We assume here that the evaluation step has knowledge of all events that have an occurrence time ending at *now*. (Variations on this where events might be delayed and arrive out of order are discussed in Section 12.2.2 and Chapter 18).

Incoming events might be events that are actually present in the event stream as messages. They might however also be absolute or relative timer events. These timer events are not present in the event stream and the query evaluation engine is responsible for "waking itself up" and generating these timer events. Note that if a timer event and a message in the event stream have occurrence times that are in close temporal proximity, then there is a potential for race conditions where the later event of the two might "overtake" the earlier. This race condition is due to the imprecision of thread and process scheduling and timers in current operating systems and programming platforms. If we assume that events are processed in an order determined by their occurrence times (cf. Section 12.2.2), then careful programming is necessary to maintain order in the presence of timer events.

12.1.2 Input and Output

For an evaluation step at time *now*, the input consists of all events with an occurrence time ending at *now*. More precisely, it is representations of the events together with occurrence times. In the case of XChange$^{\mathrm{EQ}}$, the event representations are XML messages or (equivalently) data terms. In the case of the simplified Rel$^{\mathrm{EQ}}$, the event representations are relational tuples. Occurrence times of events are time intervals $t = [b, e]$ and we have $end(t) = e = now$. Recall that events can either be from the event stream or timer events that must be "generated" by the evaluation engine.

In terms of our declarative semantics (see Chapter 10), the input is the following fragment of the event stream E: $\{e^t \mid e^t \in E,\ end(t) = now\}$.

The output of an evaluation step at time *now* in turn are the events derived by deductive rules with an occurrence time ending at *now*. Again, these events consist of a message and an

occurrence time t, where we have $end(t) = now$. (A caveat on this notion of output follows in Section 12.1.5.)

In terms of our declarative semantics, the output is the following fragment of out intended model $M_{P,E}$: $\{e^t \mid e^t \in M_{P,E},\ end(t) = now\}$. The second theorem about the declarative semantics (Chapter 11.2) ensures that this output can be computed from the event stream received so far, i.e., from $E\ |_{[0,now]} = \{e^t \mid e^t \in E,\ end(t) \leq now\}$. The time point 0 here denotes the earliest time point, i.e., the time point when the overall process of event query evaluation has been started.

As an example for the inputs and outputs of evaluation steps over time consider evaluating the following Rel$^{\mathrm{EQ}}$ program with two event query rules:

$$\begin{array}{lcl} \mathrm{c}(x) & \leftarrow & i : \mathrm{a}(x),\ j : \mathrm{b}(x),\ \{i, j\}\ \mathsf{within}\ 7 \\ \mathrm{d}(x) & \leftarrow & i : \mathrm{a},\ k : \mathsf{extend}(i, 5),\ \mathsf{while}\ k :\ \mathsf{not}\ \mathrm{b}(x) \end{array}$$

The following table shows the input and output of evaluation steps. In the input, we distinguish between events of the event stream and timer events.

eval. step at now =	1	3	4	6	8
input event stream	$\mathrm{a}(42)^{[1,1]}$	$\mathrm{a}(20)^{[2,3]}$	$\mathrm{b}(20)^{[3,4]}$	—	$\mathrm{b}(42)^{[6,8]}$
input timer events	—	—	—	$\mathsf{extend}(i,5)^{[1,6]}$	$\mathsf{extend}(i,5)^{[2,8]}$
expected output	—	—	$\mathrm{c}(20)^{[2,4]}$	$\mathrm{d}(42)^{[1,6]}$	$\mathrm{c}(42)^{[1,8]}$

When we consider only a single event query q instead of a full rule or set of rules, then there is no construction of a new event message. Accordingly, the proper output of the evaluation of the single event query q is not an event message. Instead it is the bindings for variables obtained from the query. As far as the ideas behind event query evaluation are concerned, these two types of output are however mostly interchangeable: Variable bindings might be given a generic representation as message. For example, $\Sigma = \{\{X \mapsto \texttt{"y"}\}, \{X \mapsto \texttt{"z"}\}\}$ might be represented as term `subset { subst { map["X","y"] }, subst { map["X","z"] } }`. Similarly, a message might be represented as substitution set by using an artificial variable that is bound the message. For example, using D as artificial variable, the message `a[b,c]` might be represented as $\Sigma = \{\{D \mapsto \texttt{a[b,c]}\}\}$. Importantly, both event messages and variable bindings have an associated occurrence time in event query evaluation.

12.1.3 (Partial) Event Histories

The output of the evaluation step at time now depends on more than the current input of the step. The current input is only $\{e^t \mid e^t \in E,\ end(t) = now\}$. To compute the output, however, $\{e^t \mid e^t \in E,\ end(t) \leq now\}$ is needed, i.e, also events with an occurrence time ending prior to now. These events have been the input of earlier evaluation steps.

Accordingly, evaluation steps have to preserve (some) of its current input for later steps. The data structure that is used for this is called event history. In addition to producing output, each evaluation step therefore also has to maintain the event history by updating it with the events from its current input.

The event history does not have to be a complete and explicit list or set of all events (messages and occurrence times) that have been received so far. Some information in the event history will become irrelevant over time and can be discarded (see next section). The event history therefore is only a partial history. Further the necessary information can be implicit. For example instead of storing the message of an input event (an XML document or data term), it is usually sufficient to store only the variable bindings obtained from evaluating simple event queries (query terms) against the message. The message itself is not relevant for constructing output events, only the variable bindings are.

We will see that it is desirable and common in event query evaluation to not just store records about incoming events but also to store some intermediate results (cf. Section 12.2.1). These intermediate results are also called partial answers or semi-composed events and are also stored

in the event history. These partial answers are another case where information about base events can be implicit in the event history: instead of having an explicit record about a base event, there might only be records of partial answers that were generated using that base event.

The event history is often subdivided into groups of records of events that have the same type, match the same simple event query, or have been generated by the same deductive rule. Accordingly, we will also often speak of many event histories (meaning the individual subdivisions) rather than a single event history (meaning the union of all the subdivisions).

12.1.4 Garbage Collection

The event history stores information about events that are relevant for, i.e., might contribute to, future answers. Over time, some information about events in the event history becomes irrelevant however because it cannot contribute to future answers. Garbage collection of irrelevant information in the event history is required to free up memory used by it. Since event streams are conceptually infinite, garbage collection is an important requirement in event query evaluation. Without garbage collection, the event history would grow at least linearly in the size of the event stream received so far.[1] Sooner or later we will run out of memory.

To illustrate the relevance of events and garbage collection consider again the first rule from our earlier example:

$$\mathrm{c}(x) \quad \leftarrow \quad i : \mathrm{a}(x),\ j : \mathrm{b}(x),\ \{i, j\}\ \textsf{within 7}$$

When event $\mathrm{a}(42)^{[1,1]}$ initiates an evaluation step at $now = 1$, a record of this event must be stored in the event history because a later event —such as $\mathrm{b}(42)^{[6,8]}$— might be composed with it. However, the rule has a condition that the composed events must happen within 7 time units. Our a event cannot be composed with a b event that is received at a later time point than $now = 8$ because these events would not be within 7 time units. For example it cannot be composed with $\mathrm{b}(42)^{[7,8.5]}$. Therefore the stored a event becomes irrelevant when $now > 8$ and can be garbage collected.

Garbage collection is not only necessary for records of simple events in the event history. If semi-composed events are stored in the event history, a garbage collection of these is also needed.

Garbage collection can be performed as part of evaluation steps or asynchronously in a separate execution thread. The core issue of garbage collection is not the removing irrelevant events as such, but recognizing if an event is relevant or not. In our example query, this is still fairly simple: event older than 7 time units are irrelevant. We will however see that this not always straight-forward and that semi-composed events also influence relevance.

12.1.5 Output of Event Query Programs, Refined

There is a caveat to consider with the definition of output from Section 12.1.2. So far, we have said that the output should be the events that have been derived by deductive rules. This corresponds to our intuition and is also important for debugging event query programs. However, we actually do not care that much about derived events — we care about the reactions to events. These reactions are specified with reactive rules, not deductive rules. Consider the following XChange$^{\mathrm{EQ}}$ program that has one reactive rule and two deductive rules.

[1]Because the event history also contains semi-composed events, as explained in the previous section, it might actually grow faster than linearly. For example is a semi-composed event is the cross-product of two simple events, then the event history will grow quadratically.

```
RAISE
  to(...) {
    r [var X]
  }
ON
  and {
    event i: a [var X],
    event j: b [var X]
  }
  where { i before j }
END
```

```
DETECT
  b[var X]
ON
  and {
    event k: c [var X],
    event l: d [var X]
  }
  where {k before l}
END
```

```
DETECT
  e[var X]
ON
  event k: c [var X]
END
```

With this program, the thing we are primarily interested in is sending out event `r[...]` to some recipient (specified in `to(...)`). This sending out is a reaction with a side-effect. For the output we are not really interested in derived `b[...]` and `e[...]` events since these are side-effect free and do not constitute a reaction. In fact, for that purpose the following program that has just one reactive rule would be sufficient. In this program the first deductive rule has been unfolded into the query of the reactive rule and the second deductive rule completely dropped.

```
RAISE
  to(...) {
    r [var X]
  }
ON
  and {
    event i: a [var X],
    event k: c [var X],
    event l: d [var X]
  }
  where { i before k, k before l }
END
```

It turns out however, that it is much more intuitive to think of event query evaluation in terms of deriving new events with deductive rules. Also it is important to have an evaluation method that can derive all events generated by deductive rules for a better debugging of event query programs. We therefore keep with the original notion of output for an evaluation step for most of this and the following chapters. The evaluation method that will be developed is general enough so it can be re-framed to only derive those events that are actually needed for reactions.

Note that the same dual notion of output exists for rule programs in logic programming or deductive databases. Semantics of a program, i.e., a set of deductive rules, are specified in terms of the derived facts. However, the output expected in practice are not all derived facts but the answers to a so-called goal. A goal is a query that is asked against the program and the base facts. Goals correspond to the reactive rules, or more precisely the query part of reactive rules.

Logic programming mostly uses a backward chaining evaluation, where the evaluation starts with the goal. The notion of goal is therefore integral to the operational semantics there and the two notions of output are not easily interchangeable. For event query programs, however, we usually use a forward chaining evaluation (see Section 12.2.1), where we start with base events. Therefore in evaluating event queries, the two notions of output are in a sense interchangeable. We can think of obtaining the specialized case where we only care about the output for reactive rules as a transformation of the rule program (like in the example above) or as a transformation of the query plan.

12.2 Desiderata and Design Decisions

The previous section has only explained the basic task of event query evaluation; it provides still little guidance for actually developing an event query evaluation, however. We now describe the desiderata for operational semantics that build the foundation of an efficient event query evaluation method. We also describe some of the design decisions that have been made in the operational semantics of XChange$^{\text{EQ}}$.

12.2.1 Incremental Evaluation with Intermediate Results

For efficiency reasons, it is desirable to use an incremental, data-driven evaluation method for complex event queries and rules. An incremental evaluation ensures that in an evaluation step only the required output, i.e., events with an occurrence time *now*, is produced. Events with an earlier occurrence time < *now*, that have been in the output of earlier steps, need not and will not be produced again.

Different evaluation steps often require computation of the same intermediate results. It is often desirable to store and update such intermediate results across steps to avoid recomputing them in every step. Note however that storing intermediate results consumes additional memory. Therefore, there is always a trade-off between memory usage and computation time involved.

To illustrate the importance of incremental evaluation and intermediate results, consider evaluating the following event query rule:

$$\mathrm{d}(x,y) \leftarrow i : \mathrm{a}(x),\ j : \mathrm{b}(x,y),\ k : \mathrm{c}(y).$$

A naive way to evaluate it might be to maintain sets of a, b, and c events as event histories. When ever some event happens at time *now* we perform an evaluation step. In the step we first add the new event to its corresponding history. Then we use the event histories to evaluate the event query from scratch with some traditional, non-event method. Because of the shared variables in the simple event queries, this essentially this means performing a three-way join (in the sense of relational algebra) of the sets of events stored in the event histories. The result of this join, however, contains not only our desired output of complex events with occurrence time *now* . It also contains complex events with occurrence time < *now*. We therefore need to filter the result further to obtain the desired output.

This naive method has two major issues. First, in each step we compute far more than the necessary output. We compute not only results with occurrence time *now*, but also all results with an occurrence time ≤ *now*. Only later we select the desired output from these results. In general, the number of complex events with an occurrence time < *now* can be expected to be much larger than the number with occurrence time *now*. Therefore a considerable amount of computation is wasted on producing unneeded results.

Second, each step recomputes intermediate results that have already been computed in previous steps. For example if a step is initiated by a c event, then the binary join of the event histories for a and b events is computed as an intermediate result in the three-way join. However, previous steps have done the same computation and the result has not changed because the contents of these event histories have not changed.

An incremental evaluation with intermediate results that computes only the necessary output and stores not only incoming events in the event history but also some intermediate results can be expected to perform much better. As we will see in later chapters, the term "incremental" derives from the idea that in each step we *compute only the changes* relative to the previous results given by the current incoming events. These changes, it will turn out, correspond exactly to our desired output of complex events with occurrence time *now*.

12.2.2 Timing and Order of Events

For the presentation of our operational semantics we assume that all incoming events are given occurrence times according to a single time axis. We further assume that they are received and processed by the event query evaluation engine in the order of their occurrence times. Occurrence times can be time intervals $t = [b, e]$, so more precisely we mean by this that events arrive ordered by the end time point e of that time interval. Recall that our domain of time points is linearly ordered (Chapter 9.1.2).

It is important that the assumptions of a single time axis and of an ordered arrival can be given up in the operational semantics. As discussed in Chapter 2.4.4, they might not be suited for distributed systems. Clocks in a distributed system cannot be perfectly synchronized and thus

give rise to several time axes when events are time-stamped at different nodes. Varying network latencies in event transmission give rise to an unordered arrival of events.

While we will start with these assumptions, they are not an integral part of the operational semantics that we develop in the following chapters. The operational semantics are designed so that they can be easily modified to work without these assumptions. Typically they would be replaced by an assumption of a so-called scrambling bound [MWA+03] that limits the disorder and divergence of time axes.

Still, starting with these assumptions simplifies presentation of operational semantics tremendously and make it much easier to get the general ideas across. Also in some applications, a simple solution where event are assigned time stamps upon reception and according to a local clock of the event query evaluation engine is sufficient. In this case, a single time axis and ordered arrival are given by definition. When assumptions hold, they give rise to some interesting optimizations in the query evaluation. And in turn these optimizations can often be modified to work also in cases where the assumptions do not hold.

12.2.3 (Framework for) Query Optimization

For a given event query there is not just one single way for evaluating it, there are many different ways that all achieve the same result. Such a "way" to evaluate a query is also called query plan. Different query plans for the same query differ for example in the order they perform certain operations or in the concrete data structures used for the event histories. Consider again the rule

$$\mathrm{d}(x, y) \leftarrow i : \mathrm{a}(x),\ j : \mathrm{b}(x, y),\ k : \mathrm{c}(y).$$

For example, one query plan might choose to first combine a events with b (performing something like an equi-join on the x variable) and then combine this intermediate result with c events (equi-join on y variable). Another might first combine c with b events, and then with a. One plan might use arrays for event histories, another hashes.

The performance of these different query plans will differ, in fact, might vastly differ. Performance depends highly on characteristics of the event stream and the data contained in events. Accordingly, a query plan that outperforms others on one event stream might be much worse than its alternatives on another event stream. Query optimization, that is considering different query plans for evaluating a given query and choosing one that is expected to perform well, is a vital and deeply explored issue in database systems. We can expect query optimization to be equally important for event queries.

Operational semantics for event queries should therefore not just describe one single way to evaluate a query or rule program. They should be able to capture a whole space of different query plans and provide a framework for query optimization (e.g., through rewriting query plans).

12.2.4 Soundness, Completeness, Termination

Declarative semantics for XChange$^{\mathrm{EQ}}$ have been given in the form of a model theory and associated fixpoint theory (Chapters 9 and 10). It is obvious that operational semantics should be sound and complete w.r.t. the declarative semantics. Soundness here means that any answer produced by the operational semantics is also an answer according to the declarative semantics. Completeness means the converse, anything that is an answer according to the declarative semantics will be produced by the operational semantics. Another, in the field of algorithms more common term, for soundness and completeness taken together would be (partial) correctness. Since XChange$^{\mathrm{EQ}}$ is a rule-based language the terminology from logic programming is deemed more suitable, however.

Soundness and completeness is of course more an obvious requirement than a desideratum. However, they lead to an important desideratum and are therefore listed in this section: our operational semantics should make proving soundness and completeness reasonably simple. Complicated proofs would not only be prone to contain oversights or errors (i.e., not be proofs at all); they might also be an indication that optimization (e.g., in the form of rewriting query plans) is difficult.

Event query evaluation is not a simple algorithm that just runs once. It is a step-wise procedure that involves updating and garbage collecting information in the event histories across steps. Operational semantics that can be proven sound and completeness are therefore a considerable challenge. We will see that the operational semantics of XChange^{EQ} use several intermediate representations and transformations between them to get from a given XChange^{EQ} program to the final query plan. Arguably, this chain of transformations makes the operational semantics a bit long-winded to understand at first; however is helps greatly in proving correctness: the individual transformations are all quite intuitive and their correctness is easy to see.

Along with soundness and completeness comes the question of termination of the event query evaluation. Since we work on unbounded streams, termination here means that every evaluation step terminates — not that the whole evaluation terminates (which it should *not* on infinite streams). Termination is of course desirable. However there is a trade-off between a language's expressiveness and termination. If the language is such that it allows an operational semantics that guarantees termination, then the language's expressiveness it limited (in particular it is not Turing complete). A very typical approach to this dilemma is to give an operational semantics that is, in the general case, not guaranteed to terminate, but for which subsets of the language can be identified that guarantee termination.

The operational semantics for XChange^{EQ} focus on hierarchical programs (cf. Chapter 10.1.3). For these programs termination of each evaluation step is guaranteed. However, the operational semantics can easily be extended to stratified programs (cf. Chapter 10.1.2). In this case, some programs might lead to non-terminating evaluation steps. Usually this is simply because the evaluation step would have to produce an infinite number of events as output. In Chapter 18, we resume this discussions and look at alternatives and extensions to the current operational semantics that could avoid producing such an infinite number of events.

12.2.5 Extensibility and Applicability to Other Settings

Because querying events is a young and dynamic research area, it is desirable to develop operational semantics that are extensible and applicable to other settings than just XChange^{EQ} running on a single machine.

The design of XChange^{EQ} already anticipates certain points where the language might be extended. These include: new calendric systems for generating absolute and relative timers and for expressing temporal conditions, new relationships between events such as causal or spatial relationships, enriching events with (non-event) database data, or using a different data model and query language for simple events. Our operational semantics should be able to accommodate such extensions with relative ease. In the same direction, if possible the operational semantics should be suitable not just for evaluating XChange^{EQ} programs. Ideally they should provide a common basis that could also be used for implementing other, different event query languages (e.g., composition-operator-based languages).

Further our operational semantics should not just be usable in a setting where an event query program is evaluated with a fixed query plan on a single machine. They should, for example, also provide a suitable basis for investigating adaptive query evaluation techniques that modify the query plan during its execution, distributed and peer-to-peer evaluation of event queries, or event query evaluation in mobile systems with limited connectivity, bandwidth, and computation resources.

12.3 General Ideas

We now describe the general ideas behind the operational semantics of XChange^{EQ}, which will be detailed in full in the next chapters. The focus of our operational semantics are logical query plans that describe the necessary operations in evaluating a query in a still fairly abstract way. For a given operation on the logical level (e.g., a join), there are typically many possible realizations on the physical level (e.g., nested loop join, hash join, merge join).

The logical query plans of XChange$^{\text{EQ}}$ are based on an extended and tailored variant of relational algebra called Complex Event Relational Algebra (CERA). For incremental evaluation and maintenance of event histories, algebra expressions are "differentiated" into expressions that compute only changes to event histories (this includes the output as a special case). Intermediate results are captured through so-called materialization points. For garbage collection, so-called temporal relevance conditions are derived statically at compile time from a query plan. During runtime, these conditions allow to identify events and intermediate results that have become irrelevant due to the progressing of time and can be garbage collected.

12.3.1 Relational Algebra as Foundation

A core observation in this work is that evaluation of complex event queries can be based on relational algebra and, more importantly, that we can separate the algebraic query plan and its incremental evaluation. Relational algebra is an attractive candidate for formalizing operational semantics of an event query language:

- It is a established and successful formalism in the database field. Expressiveness and complexity of relational algebra are well-understood. We can expect that proving soundness and correctness w.r.t. our declarative semantics is manageable.
- Event query evaluation can build upon a myriad of work on query optimization from databases and reconsider it in the new light of incremental evaluation on event streams. In particular this includes query rewriting and cost-based heuristics [GM93, Gra95], physical implementation of operators and index structures [Gra93], and adaptive query evaluation [DIR07].
- Relational algebra lends itself to incremental evaluation relatively easily. Issues related to that have been considered for the incremental maintenance of materialized views [GL95] and also in production rule systems [For82, Mir87].
- Optimizations that utilize temporal conditions in queries together with assumptions about the timing and arrival order of events are possible, as we will see in this work.

In our operational semantics, single XChange$^{\text{EQ}}$ or Rel$^{\text{EQ}}$ rules are translated into relational algebra expressions. Simple event queries give rise to the base relations in these expressions. Complex event queries use the a set of operators that together form a special variant of relational algebra called Complex Event Relational Algebra (CERA). This variant is particularly suited for complex event queries and restricted somewhat to make incremental evaluation easy and efficient.

For the translation of single rules into CERA, there are two important ideas: First, we "pretend" that the base relations contain all events that ever happen and (at first) ignore that expressions must eventually be evaluated in an incremental way. Of course, this is by now an "old trick" that we have used before in our declarative semantics (cf. Chapter 9). The design of CERA will ensure that the expression will still deliver the correct result up to a time point *now* when the base relations contain only events up to that particular time point *now*. Second, time stamps of events are, for most part, treated like regular data attributes. Accordingly temporal conditions will be expressed, for example, simply as selections. Allowing to treat time like data gives important flexibility and extensibility for the operational semantics. However we will make use of the special meaning of these time stamp attributes in the incremental evaluation, in optimizations, and in the garbage collection.

12.3.2 Incremental Evaluation

Simply evaluating CERA expressions from scratch in every evaluation step would be inefficient as discussed in Section 12.2.1. For a more efficient, incremental evaluation we employ a technique called finite differencing. From a given CERA expression, we will obtain a new expression that will compute only the changes to the result from changes to the base relations. The design of CERA ensures that these changes are exactly the output we require in each evaluation step.

An important issue in the incremental evaluation is indicating which intermediate results should be materialized across steps to avoid their recomputation. To this end, we will extend the algebra to query plans with materialization points. A materialization point is an equation giving the result of a CERA expression a name (similar to view expressions in databases) so that it can be used like a base relation in other CERA expressions. Materialization points correspond to event histories and thus treat storing materialized intermediate results and incoming events in a uniform way. (Accordingly, we will then often refer to intermediate results also as "events.") Further, materialization points address chaining of rules in a program when one rule accesses the results of another.

12.3.3 Relevance of Events for Garbage Collection

The incremental evaluation with materialization points addresses how new events and intermediate results are added to the event histories. It does not address how irrelevant events and intermediate results are removed from the event histories. Garbage collection is based on the idea of formalizing the notion whether an event in an event history is still relevant to the query plan at a given time point as so-called relevance conditions. These relevance conditions are evaluated at query runtime, either as part of evaluation steps or asynchronously in a separate execution step, and irrelevant events removed.

The core issue of garbage collection then is determining these relevance conditions. In this thesis we focus on a particular form of relevance, temporal relevance, which is determined from time-related conditions in queries. We develop a method for statically (at query compile time) determining temporal relevance conditions for a given query plan.

This formalization of garbage collection through relevance conditions is important in order to prove correctness of the garbage collection. We will see that this can be done in a fairly elegant way: because the relevance conditions are determined statically (at compile time) we can switch back in the proof to an "omniscient" perspective on query plans that ignores the step-wise incremental evaluation over time to some degree.

Chapter 13

Complex Event Relational Algebra (CERA)

The first building block of our operational semantics is a special variant of relational algebra called Complex Event Relational Algebra (CERA). The core idea in the design of CERA is to obtain an algebra that is expressive enough to translate XChange$^{\text{EQ}}$ rules but still restricted enough to be suitable for the incremental, step-wise evaluation that is required for complex event queries.

We explain the basic idea of using relational algebra for evaluating single event query rules focusing on the simplified event query language Rel$^{\text{EQ}}$ (Section 13.1). Some basic familiarity with relational algebra is assumed (see, e.g., [AHV95, GUW01]). We then define CERA formally (Section 13.2) and show a property of CERA called temporal preservation (Section 13.3), which is important for the incremental evaluation in the next chapter. Finally, we provide full details for the translation of XChange$^{\text{EQ}}$ rules into CERA expressions (Section 13.4).

13.1 Expressing Event Queries in Relational Algebra

To explain the basic idea of using relational algebra for event query rules, we focus first on the simplified event query language Rel$^{\text{EQ}}$: this hides the complexity of XChange$^{\text{EQ}}$, in particular with regards to constructing and querying simple events, and allows us to concentrate on the core topic of detecting complex events. For illustration, we use the following three Rel$^{\text{EQ}}$ rules, which cover all relevant aspects of querying events:

$$
\begin{array}{lll}
(1) & \text{comp}(id, p) & \leftarrow \; o : \text{order}(id, p, q),\; s : \text{shipped}(id, t),\; d : \text{delivered}(t) \\
 & & \; o \;\mathsf{before}\; s,\; s \;\mathsf{before}\; d,\; \{o, s, d\} \;\mathsf{within}\; 48 \\
(2) & \text{overdue}(id) & \leftarrow \; o : \text{order}(id, p, q),\; w : \mathsf{extend}(o, 6h), \\
 & & \; \mathsf{while}\; w :\; \mathsf{not}\; \text{shipped}(id, t),\; q < 10 \\
(3) & \text{load}(\mathsf{count}(id)) & \leftarrow \; o : \text{overdue}(oid),\; w : \mathsf{from\text{-}end\text{-}backward}(o, 24h), \\
 & & \; \mathsf{while}\; w :\; \mathsf{collect}\; \text{shipped}(id, t)
\end{array}
$$

Conceptually similar rules have been used in the use cases of XChange$^{\text{EQ}}$ in Chapter 7.1. The first rule detects completed order events as a composition of order, shipped, and delivery events. The events must happen in said temporal order within 48 hours. Variable id is the order number, p the product name, q the quantity, and t the tracking number. The second rule detects overdue orders as the absence of a shipped event in the time span of 6 hours after an order event. It applies only to orders with a quantity of less than 10 items. The third rule reports the number of shipped events that have taken place in the last 24 hours prior to an overdue event.

13.1.1 Relations for Events and Event Data

We associate a relation R_i with each simple event query $i : R(x_1, \ldots, x_n)$ in the rule body. Each event of type R that happens corresponds to one tuple in R_i. Its occurrence time interval is part of that tuple, and expressed with its starting time $i.s$ and ending time $i.e$ (where i is name of the event identifier bound in the atomic event query). Accordingly, R_i has the schema $sch(R_i) = \{i.s, i.e, x_1, \ldots, x_n\}$.

Note that we use the named perspective on relations and relational algebra here, where tuples are viewed as functions that map attribute names to values. Because variables give rise to attributes, this is more intuitive here than the unnamed perspective where attribute names are identified by their position in an ordered tuple.

Example rule (1) thus gives rise to three such relations: R_o with $sch(R_o) = \{o.s, o.e, id, p, q\}$ for $o : \text{order}(id, p, q)$, S_s with $sch(S_S) = \{s.s, s.e, id, t\}$ for $s : \text{shipped}(id, t)$, and T_d with $sch(T_d) = \{d.s, d.e, t\}$ for d : delivered(t). These relations will be the input of the relational algebra expression into which we will translate the rule.

13.1.2 Event Composition and Temporal Conditions

By virtue of representing occurrence times as part of tuples, translating the complex event query in the body of example rule (1) into a relational algebra expressions becomes quite straightforward. The combination of the three simple event queries with conjunction is expressed with natural joins. Maybe a bit surprisingly, temporal conditions (such as o before s) are expressed as selections; this works because we made temporal information (i.e., occurrence times of events) part of the data of our base relations.

In our example, we will have to join R_o, S_s, and T_d. The temporal condition o before s gives a selection with condition $o.e < s.s$, the temporal condition s before d a selection with condition $s.e < d.s$. The metric condition $\{o, s, d\}$ within 48 gives a selection with condition $\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48$.

With this, the rule body could be translated into the following relational algebra expression:

$$
\begin{aligned}
&\sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48](\\
&\quad \sigma[s.e < d.s](\\
&\quad\quad \sigma[o.e < s.s](\\
&\quad\quad\quad (R_o \bowtie S_s) \bowtie T_d))).
\end{aligned}
$$

For readability, we write parameters of operators in square brackets, e.g., $\sigma[o.e < s.s]$, rather than in the more conventional way of subscripts, e.g., $\sigma_{o.e<s.s}$.

There are of course a number of alternative relational algebra expressions that compute the same result. For example the expression

$$
\begin{aligned}
&\sigma[o.e < s.s](\\
&\quad \sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48](\\
&\quad\quad R_o \bowtie\\
&\quad\quad\quad \sigma[s.e < d.s](S_s \bowtie T_d)))
\end{aligned}
$$

would do the same as the one above. Rewriting rules could be used to transform one expression into the other. This is a well-explored topic for relational algebra and gives rise to query optimizations on the logical level such as pushing selections or reordering joins. There is also potential to simplify the relational algebra expression by reasoning about the temporal selections. For example, $\sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48]$ could be simplified to just $\sigma[d.e - o.s \leq 48]$ because of the other temporal conditions $o.e < s.s$ and $s.e < d.e$ and the implicit knowledge that $i.s \leq i.e$ for any i. The implicit knowledge $i.s \leq i.e$ for any i comes from the fact that the ending time of an event can never be before its starting time.

13.1.3 Rule Head

The expression just seen translates only the rule body. To translate the full rule, we still have to drop attributes that are not in the head (here q and t) and to generate the occurrence time of the result. Dropping attributes is simply a projection.

The occurrence time of the result will be expressed with time stamps $r.s$ and $r.e$. By definition, $r.s$ must be the smallest value of the time stamps of the input events and $r.e$ the largest (cf. Chapter 6.4 and the last line in the model theory of Figure 9.1). Here, therefore $r.s = \min\{o.s, s.s, d.s\}$ and $r.e = \max\{o.e, s.e, d.e\}$.

To generate the occurrence time of the result, we introduce an operator μ that is not part of standard relational algebra. The merging operator $\mu[j \leftarrow i_1 \sqcup \cdots \sqcup i_n](E)$ computes a new occurrence time interval (with start and end time stamps $j.s$ and $j.e$) from existing occurrence times so that it covers all these intervals, i.e., $j.s = \min\{i_1.s, \ldots i_n.s\}$ and $j.e = \max\{i_1.e, \ldots i_n.e\}$. The result contains only the new occurrence time, the input occurrence times are dropped.

Merging of time intervals is not really a new operation for relational algebra. It is equivalent to the following extended projection [GUW01], a common practical extension of relational algebra used to compute new attributes from existing ones:

$$\begin{aligned}\pi[\quad & j.s \leftarrow \min\{i_1.s, \ldots i_n.s\}, j.e \leftarrow \max\{i_1.e, \ldots i_n.e\},\\ & sch(E) \setminus \{i_1.s, \ldots i_n.s, i_1.e, \ldots i_n.e\} \quad](E).\end{aligned}$$

However, we do not want to allow arbitrary extended projections on time stamp attributes — they could violate the temporal preservation of CERA (cf. Section 13.3). Therefore we only allow its restricted use through the new μ operator.

The full rule of our example (1) then becomes the following relational algebra expression. Note that only the π and μ operator on the top have been added compared to the expression from earlier.

$$\begin{aligned}&\pi[r.s, r.e, id, p](\\ &\quad \mu[r \leftarrow o \sqcup s \sqcup d](\\ &\quad\quad \sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48](\\ &\quad\quad\quad \sigma[s.e < d.s](\\ &\quad\quad\quad\quad \sigma[o.e < s.s](\\ &\quad\quad\quad\quad\quad (R_o \bowtie S_s) \bowtie T_d))))).\end{aligned}$$

13.1.4 Relative Timer Events and Negation

To translate example rule (2), we have to accommodate two further issues: the generation of relative timer events and the negation of an event. Relative timer events are expressed as auxiliary relations that will be joined with the relations of the other events. Negation can be expressed through an anti-semi-join, or more precisely a θ-anti-semi-join that uses the θ condition for expressing the event accumulation window.

The timer event $w : \mathsf{extend}(o, 6h)$ in our example is defined relative to the event $o : \mathrm{order}(id, p, q)$, which has the corresponding relation R_o. The relation for the timer event will be denoted X_w and defined as

$$X_w := \{x \mid (x(o.s), x(o.e)) \in \pi[o.s, o.e](R_o),\ x(w.s) = x(o.s),\ x(w.e) = x(o.e) + 6\}.$$

In this definition, $x(y)$ denotes the value for attribute y of a tuple x as usual. Relation X_w contains four time stamps: the timer event's $w.s$ and $w.e$, which are computed based on the time stamps $r.s$ and $r.e$ of R, and also $r.s$ and $r.e$, which are needed for the join $R_o \bowtie X_w$. Naturally, the definition of X_w is dependent on R_o.

Recall that negation of events must still be sensitive to data. In our example rule (2), only shipped events with the same value for id as the order event are of relevance. Accordingly, an anti-semi-join is appropriate (rather than, say, a difference). In addition, negation is restricted by a time window, which is specified by another event through the while keyword. This time window can be expressed as a condition, here $w.s \leq s.s \wedge s.e \leq w.e$, where $w.s$ and $w.e$ are the time

stamps of the event giving the time window and $s.s$ and $s.e$ are the time stamps of the negated event. This condition is added to the anti-semi-join so that it becomes a θ-anti-semi-join. Recall that a θ-anti-semi-join of a relation R with a relation S can be defined in terms of other relational algebra operators as $R \overline{\ltimes}_\theta S = R \setminus \pi_{sch(R)}(\sigma_\theta(R \bowtie S))$. Here $\setminus$ is the usual difference operator of relational algebra.

Because the expression $w.s \leq s.s \wedge s.e \leq w.e$ is somewhat longwinded and we will use it fairly often, we abbreviate it also with $w \sqsupseteq s$ and write accordingly just $\overline{\ltimes}_{w \sqsupseteq s}$.[1] The intuition is that the time interval $s = [s.s, s.e]$ is a subset of $w = [w.s, w.e]$, i.e., $[s.s, s.e] \sqsubseteq [w.s, w.e]$, if and only if $w.s \leq s.s \wedge s.e \leq w.e$. To further emphasize that w is on the left hand side and s on the right hand side of the anti-semi-join, we use $\sqsupseteq$ instead of $\sqsubseteq$.

With this, example rule (2) can be expressed as the following relational algebra expression. Note that the condition on data ($q < 10$) simply becomes an ordinary (data) selection.

$$\begin{array}{l} \pi[r.s, r.e, id](\\ \quad \mu[r \leftarrow o, w](\\ \quad\quad \sigma[q < 10](\\ \quad\quad\quad (R_o \bowtie X_w) \overline{\ltimes}_{w \sqsupseteq s} S_s))), \end{array}$$

$$\text{where } X_w := \{x \mid (x(o.s), x(o.e)) \in \pi[o.s, o.e](R_o),\ x(w.s) = x(o.s),\ x(w.e) = x(o.e) + 6\}$$

There are two things to remark on this expression. First, it might seem that instead of joining with the auxiliary relation $(R_o \bowtie X_w)$ to generate the relative timer event, one might also use an extended projection in form $\pi[w.s \leftarrow o.s, w.e \leftarrow o.e + 6, sch(R_o)](R_o)$. However, this extended projection would cause difficulties in the incremental evaluation because it does not satisfy the temporal preservation of CERA. Using the auxiliary relation also gives a considerable gain in flexibility and expressivity. Novel relative timer events that cannot be expressed as simple addition or subtraction (e.g., "the next Thursday after event o") can be modeled simply as such auxiliary relations and thus integrated in the operational semantics easily. Second, one might argue that the θ-anti-semi-join is an unnecessary operation because it can be expressed using the "low-level" operations difference, projection, selection, and join. This is true, but as we will see, CERA does not allow that particular expression.[2] The reason for this is again that the operators needed for that expression one could also be used to build expression that do not satisfy the temporal preservation. Also, a θ-anti-semi-join is very valuable for an efficient incremental query evaluation.

13.1.5 Aggregation

Dealing with the event accumulation (while/collect) used to aggregate data (e.g., count) as in the example rule (3) can be broken down into two tasks. First, we have to "supply" all the data that will be aggregated. Second, we then have to actually aggregate the data. Aggregation is an operation not supported by standard relational algebra. However, the grouping operator γ is a common extension to relational algebra to support aggregation [GUW01].

Supplying the necessary data for aggregation will be done with a θ-join. As with negation and its θ-anti-semi-join, the θ condition is used to expresses the temporal window over which events are collected. For our example rule, the θ-condition is $w.s \leq s.s \wedge s.e \leq w.e$, or abbreviated $w \sqsupseteq s$, as before. However $w.s$ and $w.e$ are now from a different auxiliary relation Y because we have a different relative timer event for w. Here $Y_w := \{y \mid (y(o.s), y(o.e)) \in \pi[o.s, o.e](U_o),\ y(w.s) = y(o.e) - 24,\ y(w.e) = y(o.e)\}$, where U_o is the relation for the overdue events.

Unlike the θ-anti-semi-join $\overline{\ltimes}_\theta$, the θ-join $\bowtie_\theta$ is not really necessary. It could be just expressed with a selection σ_θ and a regular join $\bowtie$. However the θ-join is convenient because it indicates that it comes from an event accumulation rather than an event composition.

[1] Because the operator is written in infix notation, we prefer to have the θ-condition in subscript here rather than write it in square brackets as we have with prefix operators such as σ.

[2] The expression $R \setminus \pi_{sch(R)}(\sigma_\theta(R \bowtie S)$ is not allowed because CERA does not support a difference operator (cf. Section 13.2.13) and, more importantly, projections in CERA must not discard time stamps (cf. Section 13.2.7).

Because the grouping operator is an extension to relational algebra that might not be familiar to all readers it deserves some more explanation. A grouping operator $\gamma[a_1, \dots, a_n, a \leftarrow F(A)](E)$ takes as parameters a number of attribute names $\{a_1, \dots, a_n\} \in sch(E)$ ($n \geq 1$) and a single aggregation expression $a \leftarrow F(A)$. (The extension of the grouping operator to have multiple aggregation expressions is straightforward.) The aggregation expression contains an aggregation function $F(A)$ such as $COUNT(x)$ or $MAX(x)$ and x is an attribute name ($x \in sch(E)$, usually also $x \notin \{a_1, \dots, a_n\}$). The grouping operator partitions its input tuples into groups of tuples having equal values on the grouping attributes $a_1, \dots, a_n$. For each group it outputs a single tuple. Each output tuple contains the grouping attributes $a_1, \dots, a_n$ and a single aggregated attribute a built by applying the aggregation function to the group. Note that all other attributes from the input are simply dropped.

In our example rule (3), we use as aggregation attributes the time stamps $r.s$ and $r.e$ of the result and accordingly end up with $\gamma[r.s, r.e, c \leftarrow COUNT(id)]$. Taken together this gives the following relational algebra expression for example rule (3):

$$\begin{aligned}
&\gamma[r.s, r.e, c \leftarrow COUNT(id)](\\
&\quad \mu[r \leftarrow o \sqcup w \sqcup s](\\
&\qquad (U_o \bowtie Y_w) \bowtie_{w \sqsupseteq s} S_s)),
\end{aligned}$$

where $Y_w := \{y \mid (y(o.s), y(o.e)) \in \pi[o.s, o.e](U_o),\ y(w.s) = y(o.e) - 24,\ y(w.e) = y(o.e)\}$.

A naive evaluation of this expression, where each operation has a direct correspondent in the physical query plan, might be deemed rather inefficient. It would store all S_s events (right input to the θ-join) and only compute the count aggregate from these events whenever a Y_w event happens. A more efficient physical query plan might combine the join and grouping into a single operator. This combined operator could then avoid to store S_s events explicitly and only maintain aggregated counts. However, such a single operator cannot be used in all cases. It works well only on monotonous aggregates (such as $COUNT$ or MAX).[3] Further, more complicated queries might involve both negation and aggregation (and even multiple ones). In these cases the more general translation that splits collecting data (θ-join) and computing the aggregate (γ) is needed. Because we are here concerned with logical query plans (and not yet so much with the most efficient physical realization), we use the more general solution.

13.1.6 Matching and Construction in XChange$^{\text{EQ}}$

So far, we have only been looking at Rel$^{\text{EQ}}$, where an event is represented by a single tuple. Our eventual goal however are operational semantics for XChange$^{\text{EQ}}$, where events are represented as data terms not relational tuples. However, it turns out that relational algebra is still suitable for evaluating XChange$^{\text{EQ}}$. The reason for this is that the evaluation of rules is primarily concerned with variable bindings (substitutions and substitution sets).

Substitutions can be perceived as relational tuples.[4] Then, relational algebra can be used to express computations on substitutions and substitution sets. Simple event queries, which match query terms against the data terms of incoming events, produce substitution sets. Rule heads, which contain construct terms that create data terms for new derived events, consume substitution sets. The heart of complex event query evaluation are the computations and transformations done with substitutions and substitution sets on their way from simple event queries to rule heads, and for this we will use relational algebra.

To step up from Rel$^{\text{EQ}}$ to XChange$^{\text{EQ}}$, we have to

- incorporate matching of query terms to evaluate simple event queries,

[3] One might try to argue that all practically relevant aggregates are monotonous. For example, this is true for all aggregate functions supported in SQL. However, in XChange$^{\text{EQ}}$ construct terms also use a non-monotonous aggregation: the construction of a set or list of subterms with `all`. While not common for relational data, non-monotonous aggregation is common and important for restructuring XML data.

[4] The correspondence between substitutions and relational tuples is particularly emphasized here by taking the named perspective on relational algebra rather then the unnamed, positional perspective.

```
DETECT
  order {
    id { var I },
    customer { var C },
    product { var P},
    quantity { var Q },
    discount { var D }
  }
ON
  and {
    event o: weekly-offer {
      code { var C },
      product { var P },
      discount { var D },
      max-quantity { var MQ }
    },
    event a: accept-offer {
      code { var C },
      id { var I },
      customer { var C },
      quantity { var Q }
    }
  } where { o before a, {o,a} within 1 week, var Q <= var MQ }
END
```

Figure 13.1: Example XChange$^{\text{EQ}}$rule for translation into relational algebra

- incorporate construction of new data terms to evaluate rule heads, and
- deal with the technicality that matching and construction work with substitution sets not single substitutions.

For matching and construction we introduce a matching operator $\mathsf{Q}^{\mathsf{X}}[i : q]$ (or $\mathsf{Q}^{\mathsf{X}}_{i:q}$), and a construction function $\mathsf{C}^{\mathsf{X}}[c]$ (or $\mathsf{C}^{\mathsf{X}}_{c}$). Here, i is just a name giving rise to the time stamps $i.s$, $i.e$, q a query term, and c a construct term. Q^{X} is a unary operator in the algebra, C^{X} is an aggregation function (like $COUNT$ or MAX) that is used as a parameter in the grouping operator γ.

In our operational semantics Q^{X} and C^{X} realize functionality that XChange$^{\text{EQ}}$ inherits from Xcerpt. Because they are basically treated as “black boxes,” the general approach of the operational semantics XChange$^{\text{EQ}}$ might still be applicable when another underlying query language than Xcerpt is used, provided that matching and construction operations can be given for that other query language. The superscript X emphasizes that Q^{X} and C^{X} are the “Xcerpt versions.”

To group together substitutions that belong to the same substitution set (i.e., were obtained from the same incoming event), we introduce an additional attribute in tuples called event reference. An event reference $i.ref$ is basically just an identifier so that tuples with the same value belong to the same substitution set.[5] Like time stamps, event reference attributes are a kind of administrative meta-data that is used in the evaluation. They are generated by the matching operator. An alternative to event reference could be to work with relations that are in non-first normal form (NFNF), i.e., that allow nesting. However, this would lead to operational semantics that are further away from the traditional relational model and thus harder to understand and implement.

We will formally define Q^{X} and C^{X} later in Sections 13.2.11 and 13.2.12. To give a first taste however, consider the rule in Figure 13.1. The incoming event stream is modeled with a single relation E, which will be accessed by a matching operator for each simple event query. (Note that in contrast to Rel$^{\text{EQ}}$, there is no event type that would give rise to separate relations for incoming events.) The relation E has three attributes: time stamps $e.s$ and $e.e$, as we have used them before, and a data attribute $term$ for the data term of an event. With this, the rule can be translated

[5]The name “event reference” has been chosen to avoid confusion with the term event identifier already used on the language level (such as “`i`” in `event i:` q) and because one way to implement event references would be to use a memory address (i.e., a pointer or *reference*) of the event.

into the following expression of relational algebra extended with matching and construction:

$$
\begin{array}{l}
\pi[r.s, r.e, term](\\
\quad \gamma[r.s, r.e, o.ref, a.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}_{c}(I, C, P, Q, D)](\\
\quad\quad \mu[e \leftarrow o \sqcup a](\\
\quad\quad\quad \sigma[o.e < a.s](\\
\quad\quad\quad\quad \sigma[\max\{a.e, o.e\} - \min\{a.s, o.s\} \leq 1](\\
\quad\quad\quad\quad\quad \sigma[Q \leq MQ](\\
\quad\quad\quad\quad\quad\quad M[o : q_1](E) \bowtie M[a : q_2](E) \quad))))))
\end{array}
$$

In this expression c abbreviates the query term in the head of our example rule, q_1 the first simple event query in the rule body, and q_2 the second.

13.2 Formal Definition of CERA

We now formally define the operations of our Complex Event Relational Algebra (CERA). This formal definition makes clear that we use only a restricted set of relational algebra operations (e.g., no difference and union operators), have some restrictions on operators (e.g., projection must not drop time stamp attributes), and have some additional operations that are not part of the standard relational algebra (e.g., grouping or merging). After the definition, we shortly summarize these differences between CERA and traditional relational algebra. A so-incline reader may want to jump directly to this summary in Section 13.2.13 and only go back to the formal definition of CERA in case some notation or definition later in this and the next chapters cannot be grasped fully from the context.

13.2.1 Relations and Schemas

Let $AttrNames$ denote a set of possible attribute names. It is partitioned into four disjoint sets, one for names for start time stamps (form $i.s$), one for end time stamps (form $i.e$), one for event references (form $i.ref$), and one for regular data attributes (form x, not containing a dot). Let dom denote the domain, i.e., the set of all possible attribute values. Since we need to represent time stamps and data terms amongst others, $\mathbb{T} \subseteq dom$ and $DataTerms \subseteq dom$.

The basic data objects our algebra operates with are tuples under the named perspective. A named tuple t is a *partial* function $t : AttrNames \rightarrow dom$ from attribute names to attribute values. We write $r(X) = \perp$ when r is undefined for the attribute name X. Named tuples will usually be denoted with lower case letters r, s, t.

A relation R is a set of tuples. Each relation R is associated with a schema $sch(R)$. For all tuples $r \in R$ it must hold that $r(X) \neq \perp$ if $X \in sch(R)$ and $r(X) = \perp$ if $X \notin sch(R)$.

An important constraint is that every schema must contain at least one pair of time stamps, i.e., $\exists i \; \{i.s, i.e\} \subseteq sch(R)$. Time stamps occur only pairwise, i.e., $\forall i \; i.s \in sch(R) \iff i.e \in sch(R)$.

For a given schema $sch(R)$, we also introduce the following "subschemas":

- $sch_{start}(R) = \{i.s \mid i.s \in sch(R)\}$ is the set of all attribute names for start time stamps in the schema,
- $sch_{end}(R) = \{i.e \mid i.e \in sch(R)\}$ is the set of all attribute names for end time stamps in the schema,
- $sch_{time}(R) = sch_{start}(R) \cup sch_{end}(R)$ is the set of all attribute names for start or end time stamps in the schema,
- $sch_{ref}(R) = \{i.ref \mid i.ref \in sch(R)\}$ is the set of all attribute names for event references, and
- $sch_{data}(R) = sch(R) \setminus (sch_{time}(R) \cup sch_{ref}(R))$ is the set of all attribute names for regular data attributes.

The notion of a schema for a relation extends straight-forwardly to the notion of a schema for a CERA expression formed using operators. We will define it along with the operators.

We make two "sanity" assumptions about contents of relations. First, no starting time stamp in a tuple r of a relation R may be later than its corresponding end time stamp. Formally,

$$\forall r \in R.\ r(i.s) \leq r(i.e).$$

Second, all tuples with the same event reference must have the same values for the time stamps that correspond to the event reference. Formally,

$$\forall i.ref \in sch_{ref}(R) \forall r \in R \forall r' \in R.\ r(i.ref) = r'(i.ref) \Longrightarrow (r(i.s) = r'(i.s) \land r(i.e) = r'(i.e)).$$

If these sanity assumptions hold for the input relations of a CERA expression, they also hold for the result of the expression. (We don't give a formal proof of this; it is just a trivial structural induction with one case for every CERA operator.)

13.2.2 Equality and Simulation Equivalence

For values from the domain of data terms ($DataTerms$), a small but important remark is necessary. When we compare the equality of two data terms, $t_1 = t_2$, we must do this with simulation equivalence as defined in [Sch04]. This ensures that terms that are syntactically different such as `a{b,c}` and `a{c,b}` but have the same semantics in the data model are recognized as equal, i.e., `a{b,c}` = `a{c,b}` is true.

13.2.3 Selection

Selection in CERA is the same as in traditional relational algebra. For a given condition C, the selection operator takes as input a relation R and delivers as output all those tuples from R that satisfy C.

$$\sigma[C](R) = \{t \in R \mid C(t) \text{ is true}\},$$
$$sch(\sigma[C](R)) = sch(R).$$

There are no restrictions on the condition C of a selection. Typically, a condition involves a comparison operator ($=$, $<$, $>$, $\leq$, $\geq$), attribute names from $sch(R)$, and possibly constants. Importantly, the condition may operate on time stamp attributes, e.g., $C \equiv i.e < i.s$. The condition may also do some more computations, e.g., computing maxima, minima, and differences that are then used in a comparison. The typical example are conditions such as $C \equiv \max\{i.e, j.e\} - \min\{i.s, j.s\} \leq 1$ used for translating the `within` metric temporal constraint.

We note that the selection operator $\sigma[C]$ should not be confused with a substitution named σ. Fortunately, this danger is slim since in the operational semantics, the substitutions have been "replaced" by tuples in relations and will not appear anymore.

13.2.4 Renaming

The named perspective of relational algebra sometimes requires for technical reasons a renaming operator, which changes the names of attributes without affecting the result. For example, the CERA expression for rule (2) in Section 13.1 produces a relation with an attribute id. This relation is accessed in the CERA expression for rule (3) but the attribute is called oid there. (Note that simply changing oid to id is not possible, because id is already used elsewhere in rule (3) and its corresponding CERA expresssion.)

Renaming is denoted $\rho[a'_1 \leftarrow a_1, \ldots a'_n \leftarrow a_n](R)$ and renames attributes $a_1, \ldots, a_n$ respectively to $a'_1, \ldots, a'_n$. Recall from Section 13.2.1 that time stamps must always occur pairwise; accordingly, they can only be renamed pairwise.

$$\rho[a'_1 \leftarrow a_1, \dots a'_n \leftarrow a_n](R) = \{t \mid \exists r \in R.\ t(a'_i) = r(a_i) \text{ and } \forall X \notin \{a_1, \dots a_n\}\ t(X) = r(X)\},$$
$$sch(\rho[a'_1 \leftarrow a_1, \dots a'_n \leftarrow a_n](R)) = (sch(R) \setminus \{a_1, \dots, a_n\}) \cup \{a'_1, \dots, a'_n\}),$$
$$\text{where } \{a_1, \dots, a_n\} \subseteq sch(R), \{a'_1, \dots, a'_n\} \cap sch(R) = \emptyset,$$
$$\text{and } j.s \leftarrow i.s \in \{a'_1 \leftarrow a_1, \dots a'_n \leftarrow a_n\} \text{ iff } j.e \leftarrow i.e \in \{a'_1 \leftarrow a_1, \dots a'_n \leftarrow a_n\}$$

13.2.5 Natural Join

Natural join in CERA is the same as in traditional relational algebra. It combines those tuples from its input relations R and S that agree on the values of shared attributes into output tuples. Note that if $sch(R) \cap sch(S) = \emptyset$, the natural join "degenerates" to a Cartesian product.

$$\begin{aligned} R \bowtie S = \{t \mid \exists r \in R \exists s \in S \forall X.\ & \text{if } X \in sch(R) \setminus sch(S) && \text{then } t(X) = r(X), \\ & \text{if } X \in sch(S) \setminus sch(R) && \text{then } t(X) = s(X), \\ & \text{if } X \in sch(R) \cap sch(S) && \text{then } t(X) = r(X) = s(X), \\ & t(X) = \bot \text{ otherwise}\}, \\ sch(R \bowtie S) = sch(R) \cup sch(S). \end{aligned}$$

Recall from Section 13.2.2 that if $r(X)$ and $s(X)$ are data terms, then their equality in the definition means simulation equivalence. As value that is assigned to $t(X)$, either one of the two can be chosen.

13.2.6 Temporal θ-Join

Because they can be expressed as a combination of selection and natural join, θ-joins are not strictly necessary in CERA. However, the translation of while/collect in XChange^{EQ} and Rel^{EQ} is conveniently expressed with a particular θ-join. In this temporal θ-join $R \bowtie_\theta S$, the condition θ has the form $i.s \leq j.s \wedge j.e \leq i.e$, where $\{i.s, i.e\} \subseteq sch(R)$ and $\{j.s, j.e\} \subseteq sch(R)$. We also abbreviate this with $i \sqsupseteq j$ (to emphasize that $i.s$ and $i.e$ are on the left side, we write "$\sqsupseteq$" rather than "$\sqsubseteq$" with swapped arguments).

Because the temporal θ-join is expressible through other CERA operators, formal proofs about properties of CERA may ignore it.

$$R \bowtie_{i \sqsupseteq j} S = \sigma[i.s \leq j.s \wedge j.e \leq i.e](R \bowtie S).$$

In addition to providing some notational convenience, temporal θ-joins are interesting because they allow certain optimizations based on the temporal condition [BE07b].

Up until this point, the definitions of CERA operators were identical with traditional relational algebra. The operators that we will meet in the remainder of this section will diverge and be restricted some way. The restrictions all serve the purpose of achieving the temporal preservation property of CERA (see Section 13.3) that enables a step-wise and incremental evaluation as necessary for complex event queries.

13.2.7 Projection with Preservation of Time Stamp Attributes

Projection in CERA is subject to an important constraint: it must preserve all time stamp attributes and thus may only drop data attributes and event reference attributes. For a given set A of attribute names and an input relation R, where $A \subseteq sch_{data}(R) \cup sch_{ref}(R)$, it delivers as output a relation that is reduced to the schema A but otherwise equal to R.

$$\pi[A](R) = \{t \mid \exists r \in R \forall X. \text{ if } X \in A \text{ then } t(X) = r(X), \text{ otherwise } t(X) = \bot\},$$
$$sch(\pi[A](R)) = A,$$
$$\text{where } A \subseteq sch_{data}(R) \cup sch_{ref}(R).$$

The restriction that time stamps must be preserved by projection is important for the temporal preservation of CERA (see Section 13.3).

13.2.8 Merging of Time Intervals

Merging is a new operator in CERA that is not found in traditional relational algebra. The operator builds new time stamps $i.s$, $i.e$ in the output from time stamps $j_1.s, j_1.e, \dots j_n.s, j_n.e$. The new time stamps are constructed so that the interval $i = [i.s, i.e]$ covers exactly all the intervals $j_1 = [j_1.s, j_1.e], \dots, j_n = [j_n.s, j_n.e]$. Using the notation "$\sqcup$" from Chapter 9, this is written $i = j_1 \sqcup \dots \sqcup j_n$ and thus explains the notation used for the μ operator. The old time stamps $j_1.s, j_1.e, \dots j_n.s, j_n.e$ are simply dropped.

$$
\begin{aligned}
\mu[i \leftarrow j_1 \sqcup \dots \sqcup j_n](R) = \{t \mid\ & t(i.s) = \min\{r(j_1.s), \dots, r(j_n.s)\}, \\
& t(i.e) = \max\{r(j_1.e), \dots, r(j_n.e)\}, \\
& t(X) = r(X) \text{ if } X \in sch(R) \setminus \{i.s, i.e, j_1.s, j_1.e, \dots, j_n.s, j_n.e\}, \\
& t(X) = \bot \text{ otherwise}\}, \\
sch(\mu[i \leftarrow j_1 \sqcup \dots \sqcup j_n](R)) = &\ (sch(R) \setminus \{j_1.s, j_1.e, \dots, j_n.s, j_n.e\}) \cup \{i.s, i.e\}, \\
\text{where } \{j_1.s, j_1.e, \dots, j_n.s, j_n.e\} \subseteq sch(R).&
\end{aligned}
$$

As noted earlier, merging can be understood just as a restricted version of an extended projection. However, full extended projection is not allowed in CERA. It would be possible to add extended projection to CERA provided that it does not modify or drop time stamps (i.e., with the same restriction as standard projection). However, extended projection is not really necessary in CERA because the grouping operator performs the same duty (together with the construction function).

13.2.9 Temporal Anti-Semi-Join

CERA does not support arbitrary difference or anti-semi-join operations. However it does allow a special form of θ-anti-semi-joins, where the θ-condition gives a restriction so that temporal preservation (cf. Section 13.3) is assured and thus step-wise, incremental evaluation possible. The θ condition must have the form $i \sqsupseteq j$ (short for $i.s \le j.s \land j.e \le j.e$) where $i.s$, $i.e$ are some time stamp attributes of the left input relation and $j.s, j.e$ are the only time stamp attributes of the right input relation.

The temporal anti-semi-join $R \overline{\ltimes}_{i \sqsupseteq j} S$ takes as input two relations R and S, where $\{i.s, i.e\} \subseteq sch_{time}(R)$ and $\{j.s, j.e\} = sch_{time}(S)$. (Note that it is "$\subseteq$" for the time stamps of the left side of the anti-semi-join and "$=$" for time stamps on the right side!) Its output is R with those tuples r removed that have a "partner" is S, i.e., a tuple $s \in S$ that agrees on all shared attributes with r and whose time stamps $s(j.s)$, $s(j.e)$ are within the time bounds $r(i.s)$, $r(i.e)$ given by r.

$$
\begin{aligned}
R \overline{\ltimes}_{i \sqsupseteq j} S \quad &= \quad \{r \in R \mid \forall s \in S.\ \ \text{if } \forall X \in sch(R) \cap sch(S)\ r(X) = s(X) \\
& \qquad\qquad\qquad\qquad \text{then } [r(i.s), r(i.e)] \not\sqsupseteq [r(j.s), r(j.e)]\} \\
&= \quad \{r \in R \mid \forall s \in S.\ \ \exists X \in sch(R) \cap sch(S)\ r(X) \neq s(X) \\
& \qquad\qquad\qquad\qquad \text{or } r(i.s) > r(j.s) \text{ or } r(j.e) > r(i.e)\}, \\
sch(R \overline{\ltimes}_{i \sqsupseteq j} S) &= sch(R), \\
\text{where } \{i.s, i.e\} &\subseteq sch_{time}(R) \text{ and } \{j.s, j.e\} = sch_{time}(S).
\end{aligned}
$$

This definition is a somewhat length; therefore it might be easier to think of the temporal anti-semi-join as being defined as a combination of other operators:

$$R \overline{\ltimes}_{i \sqsupseteq j} S = R \setminus \pi_{sch(R)}(\sigma_{i \sqsupseteq j}(R \bowtie S)).$$

Keep in mind, however, that the expression on the right hand side of this definition is not allowed in CERA because there is no difference operator in CERA and because its projection does not preserve all time stamps.

13.2.10 Temporal Grouping

Grouping is an operator that is not part of the traditional relational algebra, but a common practical extension to it for dealing with aggregation (e.g., $COUNT$, MAX, MIN, SUM). Grouping in CERA is subject to an important restriction: all time stamps of the input relation must be used as grouping attributes. We therefore also call it temporal grouping. Again, this restriction serves to ensure temporal preservation in CERA (cf. Section 13.3).

The temporal grouping operator $\gamma[G, a \leftarrow F(A)](R)$ takes as input a relation R. Its parameters are set of attributes G, the so-called grouping attributes, and an aggregation expression. All time stamps of the input relations must be grouping attributes, i.e., $sch_{time}(R) \subseteq G$. The aggregation expression consists of an attribute name a and an aggregation function $F(A)$ with parameters A (attribute names). The grouping operator partitions R into groups P_i, one group for each combination of values of the grouping attributes G (that is, all tuples in G_i have the same values for the grouping attributes). Each group P_i gives rise to one output tuple. The output tuple contains the grouping attributes G with the corresponding values and additionally the attribute a. The value of a is obtained by applying the aggregation function $F(A)$ to P_i.

$$
\begin{array}{ll}
\gamma[G, a \leftarrow F(A)](R) = \{t \mid \exists \emptyset \neq P \subseteq R. & \forall p \in P \forall X \in G\ t(X) = p(X), \\
& \forall p' \in (R \setminus P) \exists X \in G\ t(X) \neq p'(X), \\
& g(a) = F(A)(P)\} \\
sch(\gamma[G, a \leftarrow F(A)](R)) = G \cup \{a\}, & \\
\text{where } sch_{time}(R) \subseteq G \subseteq sch(R), a \notin G \text{ is the name of a data attribute, and } A \subseteq sch(R). &
\end{array}
$$

The generalization of the grouping operator $\gamma[G, a \leftarrow F(A)](R)$ with a single aggregation expression $a \leftarrow F(A)$ to a grouping operator $\gamma[A, a_1 \leftarrow F(A_1), \ldots a_n \leftarrow F(A_n)](R)$ with multiple aggregation expressions is straightforward and will therefore not be detailed here further.

The aggregation function $F(A)$ is any function that takes as input a single relation and produces as output a single value. Common aggregation functions are $COUNT(A)$, $MAX(A)$, $MIN(A)$, and $SUM(A)$. They have the following definitions:

$$
\begin{array}{lcl}
COUNT(A)(P) & = & \mid \pi[A](P) \mid, \\
MAX(A)(P) & = & \max\{v \mid \exists p \in P.v = p(A)\}, \\
MIN(A)(P) & = & \max\{v \mid \exists p \in P.v = p(A)\}, \\
SUM(A)(P) & = & \sum_{p \in P} p(A).
\end{array}
$$

In the first case, A can be a set containing one or more attribute names. In the other three cases, A must contain only a single attribute name, and all values of that attribute in A must be numbers.

Another aggregation function is $\mathsf{C}_c^{\mathsf{X}}$, which is used for the construction of new data terms (as needed for translating XChange$^{\mathrm{EQ}}$, not just Rel$^{\mathrm{EQ}}$). This aggregation function is detailed in the next section. Because construction may actually produce not just one new data term but several, we generalize our temporal grouping operator. In the generalized version, it produces for each group not just a single tuple but a one tuple per value in the result (which is a set) of the aggregation function. Note that all that changes in the definition that follows is that we have $g(a) \in F(A)(G)$ instead of $g(a) = F(A)(G)$.

$$
\begin{array}{ll}
\gamma[G, a \leftarrow F(A)](R) = \{t \mid \exists \emptyset \neq G \subseteq R. & \forall g \in G \forall X \in G\ t(X) = g(X), \\
& \forall g' \in (R \setminus G) \exists X \in G\ t(X) \neq g(X), \\
& g(a) \in F(A)(G)\} \\
sch(\gamma[G, a \leftarrow F(A)](R)) = G \cup \{a\}, & \\
\text{where } sch_{time}(R) \subseteq G \subseteq sch(R), a \notin G \text{ is the name of a data attribute, and } A \subseteq sch(R). &
\end{array}
$$

When the aggregation functions $COUNT(A)$, $MAX(A)$, $MIN(A)$, and $SUM(A)$ are adapted so that they deliver a singleton set instead of directly delivering a value, this generalized version can be used for them as well.

13.2.11 Construction

The construction of data terms is realized as an aggregation function $\mathsf{C}^{\mathsf{X}}_c(A)$ (also written $\mathsf{C}^{\mathsf{X}}[c](A)$), where c is a construct term and X a set containing one or more attribute names. Like other aggregation functions, $\mathsf{C}^{\mathsf{X}}_c$ takes as input a relation provided by the grouping operator. As output it produces a set of data terms. These data terms are constructed from the construct term c by interpreting the input relation as a substitution set (and accordingly the tuples of the input relations as the individual substitutions of the substitution set).

$\mathsf{C}^{\mathsf{X}}_c(X)$ is more or less a black box operation defined by Xcerpt, the Web query language underlying XChange$^{\mathrm{EQ}}$. The result of $\mathsf{C}^{\mathsf{X}}_c(A)(P)$ is the application $\Sigma(c)$ of the substitution set Σ that corresponds to $\pi_A(P)$ to the construct term c as defined for Xcerpt in Chapter 7.3.3 of [Sch04]. Recall that we have met this application $\Sigma(c)$ already in Chapter 9.2.3.

$$\mathsf{C}^{\mathsf{X}}_c(X)(P) = \Sigma_{A,P}(c)$$
$$\text{where } \Sigma_{A,P} := \{\sigma \mid \exists p \in P \forall X.\ \sigma(X) = p(X) \text{ if } X \in A,\ \sigma(X) = \perp \text{ otherwise}\}$$

Keep in mind that $\mathsf{C}^{\mathsf{X}}_c(X)$ is an aggregation function (like $COUNT$ or MAX), not an algebra operator (like σ or $\bowtie$). It can only be used inside a temporal grouping expression.

Note that as already mentioned in Section 13.1.6, other construction operations, for example from other query languages than Xcerpt could be easily integrated into CERA. All we need for this is an appropriate construction function C^{L} for that language. This shows that CERA is a very general formalism for the evaluation of complex event queries that is applicable beyond Rel$^{\mathrm{EQ}}$ and XChange$^{\mathrm{EQ}}$.

13.2.12 Matching

Obtaining variables bindings by matching the query term q of a simple event query `event i:` q against data terms of events is realized by the matching operator $\mathsf{Q}^{\mathsf{X}}_{i:q}$ (also written $\mathsf{Q}^{\mathsf{X}}[i:q]$). The input of $\mathsf{Q}^{\mathsf{X}}_{i:q}$ is a relation R with schema $sch(R) = \{e.s, e.e, term\}$. The relation contains one tuple for each simple event; its starting time the value of $e.s$, its ending time the value of $e.e$ and its data term the value of $term$. The result $Q_{i:q}(R)$ is a relation that contains a tuple for each substitution obtained from matching the query term q against all the $term$ values in the input relation. These tuples have an attribute for each free variable in q, and additionally the three administrative attributes $i.s$, $i.e$, and $i.ref$, which will be detailed shortly.

Note that matching of q against a single data term yields a set of substitutions (not a single substitution). Now, $\mathsf{Q}^{\mathsf{X}}_{i:q}$ matches against all data terms in R, and each of these terms yields an individual set of substitutions. The result relation $Q_{i:q}(R)$ is however just a "flat" set of tuples for substitutions, not a set of sets. We therefore need a way to reconstruct from the flat set $Q_{i:q}(R)$ the tuples belonging together because they were obtained from the same simple event. To this end, tuples in $Q_{i:q}(R)$ contain the additional attribute $i.ref$. This attribute, called event reference, is an identifier that tells us which tuples belong together. For tuples that were obtained from the same simple event, the values of $i.ref$ are the same, for tuples obtained from different simple events they are different. There are no restrictions on the domain of the event reference attribute or how it is generated as long as it fulfills this purpose. It could be implemented for example by simply assigning consecutive numbers to simple events or as a memory address of the simple event. One could also concatenate the string representations of start time stamp, end time stamp and data term of the simple event, because the string obtained this way would be unique. (Note however that this is only interesting for theoretical investigations; the strings would consume an unnecessarily large amount of memory compared to consecutive numbers.) The generation of event references is handled by a designated function ref below.

The tuples also contain the start and end time stamp of the simple event they were obtained from. For convenience, the matching operator renames them from $e.s$, $e.e$ to $i.s$, $i.e$.

The matching itself is (like construction) a black box operation realized by Xcerpt. Below we write it as a function $match(q, d)$, where d is a data term and q is a query term that does not contain negated variables. Negated variables in query terms are those that occur

only inside a subterm negation (keyword `without`) such as the variable Y in the query term `a {{ b { var X }, without c [ var Y ] }}`. The reason that negated variables are not allowed is that they would lead to an infinite number of possible substitutions because there are always infinitely many possible bindings for the negated variable.

As detailed in work on Xcerpt [Sch04] and mentioned in Chapter 9.2.2, $match(q,d) = \{\sigma \mid \sigma(q) \preceq d, \forall X \notin FV(q)\ \sigma(X) = \bot\}$. Recall that $\preceq$ is the simulation between ground terms. $FV(q)$ denotes the free variables in q. The result of $match(q,d)$ is finite because negated variables are not allowed and we only look at substitutions σ that are defined for the free variables in q.

The matching operator $\mathsf{Q}^{\mathsf{X}}_{i:q}$ then has the following definition:

$$\begin{aligned}
\mathsf{Q}^{\mathsf{X}}_{i:q}(R) = \{t \mid \exists r \in R \exists \sigma \in match(q, r(term)).\ \ & t(i.s) = r(e.s),\ t(i.e) = r(e.e),\\
& t(i.ref) = ref(r),\\
& \forall X \in FV(q)\ t(X) = \sigma(X)\},
\end{aligned}$$
$$sch(\mathsf{Q}^{\mathsf{X}}_{i:q}(R)) = \{i.s, i.e, i.ref\} \cup FV(q),$$
where ref is a function so that for all $r, r' \in R:\ r \neq r'$ iff $ref(r) \neq ref(r')$,
and q a query term not containing negated variables.

Translation of simple event queries q containing negated variables requires some more work, which will be detailed in Section 13.4.4. Essentially, we have to split q into two query terms and use an anti-semi-join in the algebra to subtract variable bindings obtained from the two query terms.

As with the construction function, it is easy to extend CERA to incorporate matching operations from other query languages than Xcerpt. All it needs is a new operator Q^{L} that is defined similar to Q^{X} but for example with a different matching function *match*.

Note that we require only the *matching* between a query term and a data term in our algebra. A full *unification* between a query term and a construct term is neither supported nor necessary, since we aim at a forward-chaining evaluation. Unification of terms as it would be required by a backward-chaining evaluation has been developed for Xcerpt and is described in [Sch04].

13.2.13 Summary of Differences to Traditional Relational Algebra

Since CERA is a variant of relational algebra, it is helpful to highlight its differences with the traditional model. The first difference it that we use some administrative attributes in addition to regular data attributes in CERA:

- All relations must have at least one pair $\{i.s, i.e\}$ of time stamps attributes.
- Some relations have event reference attributes such as $i.ref$.
- There are "sanity" assumptions about these administrative attributes, namely $\forall r \in R.\ r(i.s) \leq r(i.e)$ and $\forall r \in R \forall r' \in R.\ r(i.ref) = r'(i.ref) \Longrightarrow r(i.s) = r'(i.s) \wedge r(i.e) = r'(i.e)$.

The following operators in CERA are the same as in traditional relational algebra and have no special restrictions:

- selection σ,
- renaming ρ, and
- natural join $\bowtie$.

The following operators in CERA are the same as in traditional relational algebra (with common extensions such as grouping), but have some special restrictions:

- Projection π must not drop time stamp attributes.

- Anti-Semi-Joins $\overline{\ltimes}$ must have a θ-condition of the form $i \sqsupseteq j$ where i is an occurrence time of the left relation and j the only occurrence time of the right input relation.
- Grouping γ must use all time stamp attributes of its input relation as grouping attributes.

The following CERA operators are new and not part of traditional relational algebra:

- Merging μ creates a new time interval (as a time stamp attribute pair) by merging and discarding time intervals (given by several time stamp attribute pairs) from each input tuple. Note that merging could also be seen as a restricted version of extended projection.
- Matching Q^{X} "converts" data terms (represented as a single attribute *term*) into variable bindings (represented several attributes corresponding to the variable names). Additionally, matching equips its result tuples with time stamp and event reference attributes.
- Construction C^{X}, which is not really an operator but an aggregation function, "converts" tuples representing variable bindings back into data terms.

The following operators are not part of CERA, although they can be found in traditional relational algebra:

- difference $\setminus$, and
- union $\cup$.

A difference operation is simply not necessary in CERA, because we already have the (restricted) anti-semi-join. Union has not been included in CERA for cosmetic reasons that have to do mostly with the presentation of the algorithm for determining temporal relevance (see Chapter 15). Union could easily be added to CERA, because the temporal preservation property (cf. Section 13.3) would still hold. For the translation of rules, union is not needed in CERA because we can split rules that contain disjunction into several rules. However, the query plans for evaluating full programs (not just single rules) that will be presented in next chapter support some form of union.

As mentioned in the beginning of this chapter, the core idea in the design of CERA is to obtain an algebra that is

- expressive enough to translate XChange$^{\text{EQ}}$ rules (entailing a need for operations such as grouping, matching, and construction), but still
- restricted enough to be suitable for the incremental, step-wise evaluation that is required to evaluate complex event queries.

13.3 Temporal Preservation in CERA

CERA expressions pretend a kind of "omniscience" that we will not have in the actual query evaluation, which must be done over time in a step-wise manner as events are received: the relations contain conceptually *all* events that ever happen. In the actual evaluation of event queries, on the other hand, we can work only with histories of events and have no way of knowing future events.

The restrictions CERA imposes on expressions (as compared to an unrestricted relational algebra), make a step-wise evaluation over time reasonable since we do not need any knowledge about future events when we want to obtain all results of an expression with an occurrence time until *now*. More precisely, to compute all results of a CERA expression E with an occurrence time before or at time point *now*, we need to know its input relations only up to this time point *now*.

This property called the temporal preservation of CERA, because when we replace the "omniscient" input relations of an expression E with histories up to *now*, then all results with an occurrence time up to *now* are preserved. To formally state this property, we define the term "occurrence time" of an event tuple and introduce two related shorthand notations for convenience.

Definition The occurrence time of a tuple e in the result of an expression E is the latest time stamp in e, i.e., $m_E(e) := \max\{e(i.e) \mid i.e \in sch_{end}(E)\}$.

The shorthand $m_E(e)$ is introduced because we will need this longwinded expression fairly often. To refer to the occurrence time in selections we also introduce the shorthand $M_E := \max\{i_1.e, \ldots, i_n.e\}$ where $\{i_1.e, \ldots, i_n.e\} = sch_{end}(E)$. M_E is basically the same as m_e, only on the syntactic level of the algebra (as needed in conditions of selections) instead of the semantic level of individual tuples.

Theorem Let E be an CERA-expression with input relations $R_1, \ldots, R_n$. Then for all time points *now*: $\sigma[M_E \leq now](E) = E'$, where E' is obtained from E be replacing each R_k with $R'_k := \{r \in R_k \mid m_{R_k}(r) \leq now\}$. We call this the temporal preservation property of CERA.

Proof By design of CERA, each operator is restricted in such a way that it maintains the temporal preservation property. Therefore the proof is a simple structural induction with cases for each operator; it is sketched in Appendix B.1.

Counter-Example Consider the following relational algebra expression E that has two input relations R and S for events with $sch(R) = \{i.s, i.e, x\}$ and $sch(R) = \{j.s, j.e, x\}$:

$$E = R \mathbin{\overline{\ltimes}} \pi[x](S).$$

This expression is *not* a CERA expression because it uses an unrestricted anti-semi-join and a projection that discards time stamps. The expression does not satisfy temporal preservation. That is, we cannot compute $\sigma[M_E \leq now]$ from $R' := \{r \in R \mid m_R(r) \leq now\}$ and $S' := \{s \in S \mid m_S(s) \leq now\}$. In fact to compute $\sigma[M_E \leq now]$, we have to know the full relation S instead of just its history up until the time *now*. Therefore, E cannot be evaluated in the step-wise manner over time that is associated with event queries.

This general idea behind this theorem about our operational semantics is similar to the idea behind the second theorem about our declarative semantics (see Chapter 11.2). Both theorems show that the suitability of the semantics for complex event queries that must be evaluated over time in a step-wise manner. Apart from one being about operational and the other about declarative semantics, there is another important difference between the two theorems. The theorem about operational semantics given here is concerned with deriving *all* events for a single rule. (In the next chapter, we will see that temporal preservation extends beyond CERA expressions for single rules to query plans for full programs.) The theorem about declarative semantics is concerned with checking the entailment of a *single* event under a full program.[6]

13.4 Translation of single XChange$^{\mathbf{EQ}}$ rules into CERA

In Section 13.1, we have explained the basic idea for translating Rel^{EQ} and XChange^{EQ} rules into CERA expressions. We have however not provided full details or an indication that the translation is correct with respect to our declarative semantics. The translation of single XChange^{EQ} rules into CERA expression is now the topic of this section.

13.4.1 Normalization of Rules

The translation of rules is more convenient if we consider them only in a normalized form. In the normal form, the rule body contains a single multi-ary conjunction (i.e., a single `and`). Further, the literals b_i in the rule body must be in an order so that simple event queries come first, relative timer events next, then while/collect literals, and at the end while/not literals. Relative timer events

[6]From this it follows that the temporal preservation theorem here only has an "upper bound" *now*, while the theorem for declarative semantics has both an "upper" and "lower bound" given by the time interval t in it.

```
DETECT
  h
ON
  and {
      b_1,
      ...,              b_i ≡ event j_i : q_i   (1 ≤ i ≤ k)
      b_k,
      b_{k+1},
      ...,              b_i ≡ event j_i : relative-timer-spec_i[j_{a_i}, d_i]   (1 ≤ i ≤ k; a_i < i)
      b_l,
      b_{l+1},
      ...,              b_i ≡ while j_i : collect q_i   (l + 1 ≤ i ≤ m)
      b_m,
      b_{m+1},
      ...,              b_i ≡ while j_i : not q_i   (m + 1 ≤ i ≤ n)
      b_n,
  } where { c_1 , ..., c_p }
END
```

Figure 13.2: Normal form for rules

$$\frac{h \leftarrow \texttt{or}\{q_1, \ldots, q_n\}\ \texttt{where}\ C}{h \leftarrow q_1\ \texttt{where}\ C \quad \vdots \quad h \leftarrow q_n\ \texttt{where}\ C}$$

$$\frac{h \leftarrow \texttt{and}\{q_1, \ldots,\ \texttt{or}\{q'_1, \ldots q'_m\}, \ldots q_n\}\ \texttt{where}\ C}{h \leftarrow \texttt{and}\{q_1, \ldots, q'_1, \ldots, q_n\}\ \texttt{where}\ C \quad \vdots \quad h \leftarrow \texttt{and}\{q_1, \ldots, q'_m, \ldots, q_n\}\ \texttt{where}\ C}$$

$$\frac{h \leftarrow \texttt{and}\{q_1, \ldots,\ \texttt{and}\{q'_1, \ldots q'_m\}, \ldots q_n\}\ \texttt{where}\ C}{h \leftarrow \texttt{and}\{q_1, \ldots, q'_1, \ldots, q'_m, \ldots, q_n\}\ \texttt{where}\ C}$$

$$\frac{h \leftarrow q \qquad q \text{ is a simple event query}}{h \leftarrow \texttt{and}\{q\}}$$

Figure 13.3: Rewriting rules for obtaining a set of normalized rules

must be ordered so that the event identifier that is used in the definition of a timer event (e.g., i in `event` i : `extend`$[i, 5]$) has been defined previously (e.g., with an `event` i : `extend-begin`$[o, 3]$).

The structure of a rule in normal form is depicted in Figure 13.2.

In the normal form, query terms must not contain optional or negated variables (i.e., variables that occur only inside Xcerpt's `optional` or `without`). Optional variables can be dealt with by splitting a rule into two rules (similar to disjunction). Negated variables are a bit more tricky. Both topics are addressed in Section 13.4.4.

Given an arbitrary XChange$^{\text{EQ}}$ rule, we can translate it into a set of rules in normal form with a few rewriting rules that are shown in Figure 13.3. These rewriting rules eliminate disjunction (`or`) by splitting up rules. Additionally, they remove unnecessary nesting of `and`. That rewriting a set of XChange$^{\text{EQ}}$ rules with them does not affect semantics, terminates, and is confluent (i.e., the final result does not depend on the order in which the rewriting rules are applied) should be clear since the rewriting rules are just standard transformations.

When the rewriting rules have been applied, we only have to reorder the literals in each rule so that we obtain rules in normal form. Within the relative timer events, temporal stratification ensures that we actually can order them appropriately because it forbids cyclic definitions of relative timers (cf. Chapter 10.1)

A normal form for Rel$^{\text{EQ}}$ rules would be defined similarly. Since Rel$^{\text{EQ}}$ rules have no disjunction, only reordering of the literal is needed for their normal form there so that, as in XChange$^{\text{EQ}}$ simple event queries come first, then relative timer event literals, then while/collect literals, and finally while/not literals.

13.4.2 Translation of Normalized Rules

We now turn to the translation of a single XChange^{EQ} rule that has the described normal form into a CERA expression. Recall that the normalization of a single rule usually yields a set of rules not a single rule. The translation of rule sets requires additionally the notion of query plans, which will be introduced in the next chapter. While we focus here on the translation of XChange^{EQ}, the translation of Rel^{EQ} would be analogous, just without matching and construction operators.

The translation is mostly concerned with the conjunction in the rule body; recall that is has the form $\texttt{and}\{b_1, \ldots, b_n\}$. We will translate it with a series $B_1, \ldots, B_n$ of CERA expressions, where each B_i translates the fragment of the rule body consisting of the literals up i, i.e., translates $\texttt{and}\{b_1, \ldots, b_i\}$, and depends on the previous one (except of course the first). The last one, B_n, is the translation of the full conjunction in the rule body. Based on this B_n, we will then define an expression C that translates the full rule body, i.e., "adds" the conditions of the `where`-clause. Based on C, we then finally define an expression Q that translates the full rules, i.e., "adds" the rule head.

In the following, we consider a single normalized XChange^{EQ} rule which we want to translate. We use the notation from Figure 13.2; in particular, k denotes the position of the last literal that is a simple event query, l the position of the last literal that is a relative timer event, m the position of the last while/collect literal, and n the total number of literals.[7] Further let E be a relation that denotes the stream of incoming events with $sch(E) = \{e.s, e.e, term\}$ (cf. Section 13.1.6). In the translation we ignore absolute timer events because, as described in Chapter 9.4.3, they can be translated like simple event queries, only against a different stream of incoming events that represents the "calendar."

Single Simple Event Query (B_1)

The CERA expression B_1 translates only the literal b_1. This literal must be a simple event query, i.e., $b_1 \equiv \texttt{event} j_1 : q_1$.[8] We translate b_1 with the matching operator for the query term q_1. Its input is the stream of incoming events E.

$$B_1 = \mathsf{Q}^{\mathsf{X}}[j_1 : q_1](E)$$

Further Simple Event Queries (B_i, $1 < i \leq k$)

For $i > 1$, B_i translates the conjunction $\texttt{and}\{b_1, \ldots, b_i\}$. We have to distinguish four cases, depending on what kind of literal b_i is. The first case is that $1 < i \leq k$, meaning that b_i is a simple event query, $b_i \equiv \texttt{event} j_i : q_i$. We translate the single literal b_i itself as described above with a matching operator for q_i. This is then joined with B_{i-1}, the translation of $\texttt{and}\{b_1, \ldots, b_{i-1}\}$.

$$B_i := B_{i-1} \bowtie \mathsf{Q}^{\mathsf{X}}[j_i : q_i](E) \quad (1 < i \leq k)$$

Relative Timer Events (B_i, $k < i \leq l$)

The second case is that b_i is a relative timer event, `event` j_i : *relative-timer-spec*$_i[j, d_i]$. We translate b_i with an auxiliary relation X_{j_i} and join it with B_{i-1}.

$$B_i = B_{i-1} \bowtie X_{j_i}$$

The definitions of the auxiliary relations are straightforward and according to their definitions in the model theory of Figure 9.2. For example `event` $j_i : \texttt{extend}[j, 6]$ has the following auxiliary relation X_{j_i}, provided that the literal that defines j is not in turn itself a relative timer:

$$X_{j_i} := \{x \mid \quad (x(j.s), x(j.e)) \in \rho[j.s \leftarrow e.s, j.e \leftarrow e.e](\pi[e.s, e.e](E)), \\ x(i.s) = x(j.s),\ x(i.e) = x(j.e) + 6\}.$$

[7] If there are no relative timer event literals, then $l := k$. If there are no while/collect literals, then $m := l$.

[8] Note that b_1 cannot be a while/collect or while/not literal, because the event identifier after `while` must be defined earlier somewhere in the rule body. Further b_1 cannot be a relative timer event, because of temporal stratification.

In this translation, we define X_{j_i} relative to the stream of incoming events E. Alternatively, we could also define it relative to the expression used for translating the literal that defines i. (See also Chapter 14.4.4)

This alternative is mainly interesting for a more efficient implementation. If the literal defining j is also a relative timer event, then we have to use the auxiliary relation X_j of j instead of E in the definition of X_{j_i}:

$$X_{j_i} := \{x \mid \quad (x(j.s), x(j.e)) \in \pi[j.s, j.e](X_j), \\ x(i.s) = x(j.s),\; x(i.e) = x(j.e) + 6\}$$

Event Accumulation for Collection (B_i, $l < i \leq m$)

The third case is that $l < i \leq m$, meaning that $b_i \equiv$ `while` j_i : `collect` q_i (q_i query term) is a while/collect literal. The query term q_i is translated with a matching operator $\mathsf{Q}^{\mathsf{X}}[i' : q_i]$, where i' is fresh event identifier (i.e., one that is not used elsewhere in the rule or its translation). A temporal join combines this with B_{i-1} and is followed by a projection that discards $i'.ref$ (which will not be needed anymore).

$$B_i := \pi[Sch \setminus i'.ref](B_{i-1} \bowtie_{j_i \sqsupseteq i'} \mathsf{Q}^{\mathsf{X}}[i' : q_i](E)) \quad (l < i \leq m)$$
where $Sch := sch(B_{i-1} \bowtie_{j_i \sqsupseteq i'} \mathsf{Q}^{\mathsf{X}}[i' : q](E))$
and i' is a fresh event identifier

Event Accumulation for Negation (B_i, $m < i \leq n$)

The last case for the translation of the conjunction is that $m < i \leq n$, meaning that $b_i \equiv$ `while` j_i : `not` q_i (q_i query term) is a while/not literal. The query term q_i is again translated with a matching operator $\mathsf{Q}^{\mathsf{X}}[i' : q_i]$, where i' is a fresh event identifier. A temporal anti-semi-join is used to combine it with B_{i-1}.

$$B_i := B_{i-1} \overline{\ltimes}_{j_i \sqsupseteq i'} \mathsf{Q}^{\mathsf{X}}[i' : q_i](E) \quad (m < i \leq n)$$
where i' is a fresh event identifier

Conditions (C)

To translate the full rule body (i.e., the conjunction together with the `where`-condition), we add selections for the individual conditions $c_1, \ldots c_q$ "on top" of B_n.

$$C := \sigma[c'_1 \wedge \cdots \wedge c'_q](B_n)$$

Each individual condition c_i on the level of the language XChange$^{\mathrm{EQ}}$ must be transformed into a condition on the level of the algebra. For example $c \equiv i$ `before` j would become $c' \equiv i.e < i.s$. Figure 13.4 shows the translations of the temporal conditions (compare this also with Figure 9.3 defining the semantics of the temporal conditions in the model theory).

If the rule contains no `where` clause, then of course just $C := B_n$. Note that instead of a single selection with a conjunction, we could also chain multiple selections in the manner of $\sigma[c'_q](\ldots \sigma[c'_1](B_n) \ldots)$.

Full Rule (Q)

For the translation of the full rule, it remains to add the translation of the head. This means adding a merge operator that generates the time stamps $r.s$ and $r.e$ of the resulting events, a construction that generates their data term $term$, and finally a projection that discard the event reference attributes that should not be part of the output.

c (XChange^{EQ} syntax)	c' (Algebra)
i `before` j	$i.e < j.s$
i `after` j	$j.e < i.e$
i `during` j	$j.s < i.s \wedge i.e < j.s$
i `contains` j	$i.s < j.s \wedge j.e < i.s$
i `overlaps` j	$j.s < i.s \wedge i.s < j.e \wedge j.e < i.e$
i `overlapped-by` j	$i.s < j.s \wedge j.s < i.e \wedge i.e < j.e$
i `meets` j	$i.e = j.s$
i `met-by` j	$j.e = i.s$
i `starts` j	$i.s = j.s \wedge i.e < j.e$
i `started-by` j	$j.s = i.s \wedge j.e < i.e$
i `finishes` j	$j.s < i.s \wedge i.e = j.e$
i `finished-by` j	$i.s < j.s \wedge j.e = i.e$
i `equals` j	$i.s = j.s \wedge i.e = j.e$
$\{i_1, \ldots, i_n\}$ `within` d	$\max\{i_1.e, \ldots i_n.e\} - \min\{i_1.s, \ldots i_n.s\} \leq d$
$\{i, j\}$ `apart-by` d	$\min\{i.e, j.e\} - \max\{i.s, j.s\} \geq d$ or equivalently $(i.e - j.s \geq d \vee j.e - i.s \geq d)$

Figure 13.4: Translation of conditions in `where` clause

$$
\begin{aligned}
Q \quad := \quad & \pi[r.s, r.e, term](\\
& \quad \gamma[r.s, r.e, Sch_{ref}, term \leftarrow \mathsf{C}^{\mathsf{X}}_{h}(Sch_{data})](\\
& \quad\quad \mu[r \leftarrow j_1 \sqcup \cdots \sqcup j_l](C))) \\
\text{where } & Sch_{ref} = sch_{ref}(C),\ Sch_{data} = sch_{data}(C)
\end{aligned}
$$

Summary

To summarize the translation process, the general structure of the resulting CERA expression Q is illustrated in Figure 13.5. The "heart" of Q is a tree of joins, which from left to right are first regular natural joins for the simple event queries, then regular natural joins for the relative timer events, then temporal joins (together with projection to discard the event references) for the while/collect literals, and finally temporal anti-semi-joins for while/not literals. On top of this join tree sits a selection for the conditions of the `where` clause, followed by a merge operator to generate the occurrence time of the result, and a temporal grouping operator that produces the data terms of the result events.

Keep in mind that the illustration of Figure 13.5 describes the most general case, and thus gives a fairly huge and complex CERA expression. In practice, rules will rarely use all features together so that their CERA expressions will be smaller and look less complex.

13.4.3 Correctness

We now want show the correctness of the translation of a normalized rule $h \leftarrow B$. We give the main ideas here; the proof is completed in Appendix B.2.

First it is appropriate to recall some details from the declarative semantics to explain what correctness here means. The semantics are given by $M_{\{h \leftarrow B\},E}$, where $\{h \leftarrow B\}$ is the program containing only the single rule $h \leftarrow B$. Since we consider only hierarchical programs and only a single rule, we have $M_{\{h \leftarrow B\},E} = T^{\omega}_{\{h \leftarrow B\}}(E) = T_{\{h \leftarrow B\}}(E)$ so that the semantics are given by applying the fixpoint operator to the program $\{h \leftarrow B\}$ and the event stream E once. That is, the declarative semantics of a single rule $h \leftarrow B$ are given by:

$$
\begin{aligned}
T_{\{h \leftarrow B\}}(E) = E \cup \{e^t \mid \quad & \text{there exists a maximal substitution set } \Sigma, \\
& \text{and a substitution } \tau \text{ such that } E, \Sigma, \tau \models B^t \text{ and } e \in \Sigma(h)\}
\end{aligned}
$$

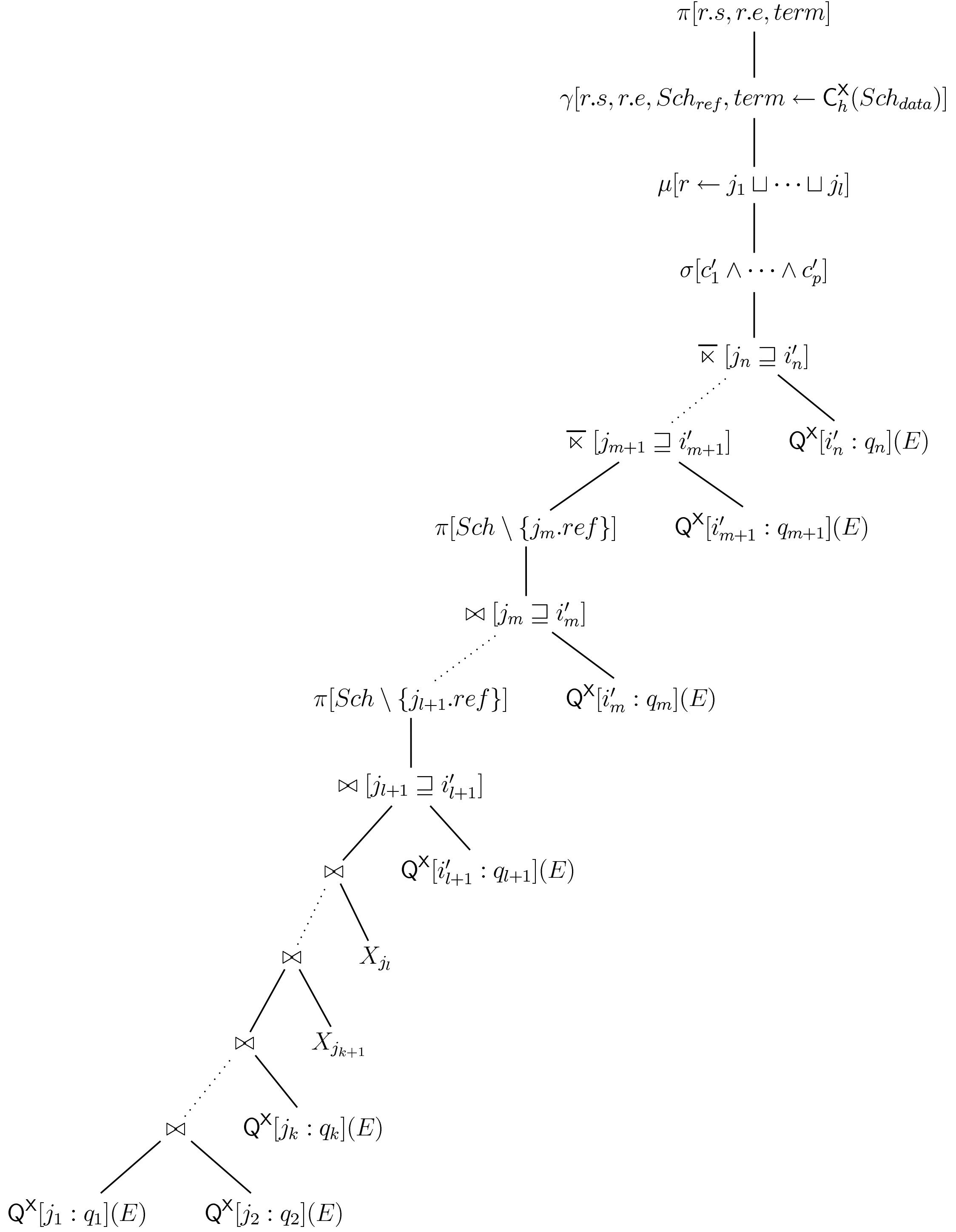


Figure 13.5: General structure of CERA expression for translation of a normalized rule

Let Q be the CERA expression that translates of rule $h \leftarrow B$. We identify events e^t with tuples r of Q and E by $r(r.s) = begin(t)$, $r(r.e) = end(t)$, $r(term) = e$. In the other direction we identify tuples r with events $r(term)^{[}r(r.s), r(r.e)]$. (Analogous for $e.s$, $e.e$ instead of $r.s$, $r.e$.) We then have to show for the correctness of our translation that[9]

$$Q \cup \rho[r.s \leftarrow e.s, r.e \leftarrow s.e](E) = T_{\{h \leftarrow B\}}(E).$$

For this it suffices to show that

$$Q = \{e^t \mid \text{ there exists a maximal substitution set } \Sigma, \text{ and a substitution } \tau \text{ such that } E, \Sigma, \tau \models B^t \text{ and } e \in \Sigma(h)\}.$$

Next, we have to find a correspondence between the contents of the relations generated by subexpressions of Q and the combined Σ and τ. This correspondence is given by "bundling together" tuples in a relation with the same values for the event references. Each "bundle" corresponds to a Σ, τ combination. The data attributes of the tuples in a bundle correspond to the individual substitutions in Σ and the event references and time stamps (which are equal for all tuples in a bundle!) to the information in τ.

We use this correspondence to show a lemma about each subexpression S of Q that translates a subexpression F of the rule body B. The subexpressions S of Q are $B_1, \ldots, B_n$, C and they translate $b_1, \ldots,$ `and` $\{b_1, \ldots b_n\}$, `and` $\{b_1, \ldots b_n\}$ `where` $\{c_1, \ldots c_p\}$, respectively. Explanations on notation following shortly, the lemma is:

$$\begin{array}{ll} \text{if} & S' \subseteq_{\text{bundle}} S, t = occtime(S') \\ \text{then} & E, \Sigma_{S'}, \tau_{S'} \models F^t, \Sigma_{S'} \text{ maximal w.r.t. } FV(F) \end{array}$$

and, conversely,

$$\begin{array}{ll} \text{if} & E, \Sigma, \tau \models F^t, \Sigma \text{ maximal w.r.t. } FV(F) \\ \text{then} & S'_{\Sigma,\tau} \subseteq S, t = occtime(S'). \end{array}$$

The first part of the lemma is soundness ("results produced by the operational semantics are results according to the declarative semantics"). The second part is completeness ("results according to the declarative semantics are actually produced by the operational semantics").

The following notation is used in this lemma. $S' \subseteq_{\text{bundle}} S$ means that S' is a maximal subset of S s.t. the values of its tuples for their event reference attributes are all equal.

$$\begin{array}{rll} S' \subseteq_{\text{bundle}} S :\iff S' \neq \emptyset \wedge \forall s' \in S'. & & \forall s'' \in S' \forall x (s'(x.ref) = s''(x.ref)) \\ & \wedge & \forall s \in S \setminus S' \exists x (s'(x.ref) \neq s(x.ref)) \end{array}$$

The occurrence time of a bundle S' is a time interval given by

$$occtime(S') = [\min\{s'(x.s) \mid x.ref \in sch_{ref}(S')\}, \max\{s'(x.e) \mid x.ref \in sch_{ref}(S')\}],$$
where s' some arbitrary tuple of S'

Note that the choice of $s' \in S'$ does not matter due to the sanity assumptions about S. (Tuples with the same value for the event references have the same values for the time stamps.)

$\Sigma_{S'}$ and $\tau_{S'}$ are the substitution set and event trace corresponding to S' as described informally earlier. Similarly, $S'_{\Sigma,\tau}$ is the subset of S (the "bundle") corresponding to Σ and τ. Their formal definitions are given in Appendix B.2.

The proof of the lemma is shows by induction that it holds for each B_i (as defined in the previous section), and then additionally that is then also holds for C. The details are given in Appendix B.2.

Knowing that the lemma holds for C, it is easy to reason that Q is correct. C represents (with the correspondence of the lemma) all possible Σ, τ combinations. The merging operator of Q generates the appropriate occurrence time $[r.s, r.e]$ for each bundle $C' \subseteq_{\text{bundle}} C$) because of

[9]The renaming (ρ) is just a minor technical matter that is needed because we the time stamps in the E are named $e.s$ and $e.e$, and thus must thus be renamed to $r.s$ and $r.e$ for the union.

the part about $occtime(C) = t$ in the lemma. The following grouping operator γ forms groups that correspond exactly to the bundles C' of C, because all event reference attributes are grouping attributes. The generated data term for each bundle is correct simply by definition of $\mathsf{C}^{\mathsf{X}}_h$. The final projection just discards the event reference attributes that should not be part of the output.

13.4.4 Incorporating Negated and Optional Variables

So far we have made the assumption that query terms do not contain negated or optional variables as defined by Xcerpt. Negated variables are variables that occur inside a subterm negation `without`, optional variables are variables that occur inside an optional subterm specification `optional`. We refer to [Sch04] for details.

Subterm negation and optional subterm specifications are important features of Xcerpt for making incomplete queries against Web data (i.e., queries that do not rely on a strict schema of the data). However, this thesis focuses on complex event queries, only building upon Xcerpt. Therefore, we shortly sketch here how negated and optional variables can be incorporated in the operational semantics developed here without providing much detail.

When we match a query term such as $q =$ `a{{without var X}}` against a data term such as `a{1,2,3}`, then we need some way to capture the information that X may never be bound to bindings `1`, `2`, or `3`. This is important for cases where the variable X is used in another query term of the same event query. In our operational semantics, the result of matching (i.e., of the matching operator Q^{X}) is a relation and tuples in the relation cannot capture such negative information. Consider the following example XChange$^{\text{EQ}}$ query:

```
and {
  event i: a {{ without var X }},
  event j: b {{ var X }}
}
```

For events $\texttt{a\{1,2,3\}}^{t_a}$ and $\texttt{b\{2,4\}}^{t_b}$, it should yield only the variable binding $X = 4$. However, the CERA expression

$$\mathsf{Q}^{\mathsf{X}}[i : \texttt{a\{\{without var X\}\}}] \bowtie \mathsf{Q}^{\mathsf{X}}[j : \texttt{b\{\{var X\}\}}]$$

would also generate $X = 2$ because the result of $\mathsf{Q}^{\mathsf{X}}[i : \texttt{a\{\{without var X\}\}}]$ does not capture that X must not be 1, 2, or 3.

One way to incorporate negative bindings would be to allow tuples r to represent negative information, here for example by having tuples that have something like $r(X) = \neg\{1, 2, 3\}$. The equality used in joins must then be adapted appropriately so that a tuple s with $s(X) = 2$ will not join with r, but a tuple s with $s(X) = 4$ will. However, because that solution would be a severe "intrusion" into the algebra it is not considered further here.

Another, simpler and cleaner possibility is to split up the query term. In the example, the query term would be split into two positive queries `a{{ }}` and `a{{var X}}`. We can then use an anti-semi-join to subtract the "forbidden" bindings from those obtained from the positive bindings. The corresponding CERA expression for our example is:

$$(\mathsf{Q}^{\mathsf{X}}[i : \texttt{a\{\{\}\}}] \bowtie \mathsf{Q}^{\mathsf{X}}[j : \texttt{b\{\{var X\}\}}]) \,\overline{\ltimes}_{i.ref=i'.ref}\, \mathsf{Q}^{\mathsf{X}}[i' : \texttt{a\{\{var X\}\}}]$$

It is important that the anti-semi-join comes after the join of the other two matchings. The operation $R \,\overline{\ltimes}_{i.ref=i'.ref}\, S$ has not been defined in CERA earlier, but is just a variant of the temporal anti-semi-join because equal event references imply equal time stamps (i.e., $i \sqsupseteq i'$ is implied by $i.ref = i'.ref$).

The situation for optional variables is similar. When we match a query term such as `a{{ optional b { var X } }}` against data terms, then X may or may not have a binding. One way to deal with this is to represent non-existing bindings with null-values and modify the equality of the joins appropriately. Another is to split the rule into one rule where the `optional` and

its subterm is removed and another rule where only the keyword `optional` is removed but the subterm stays. These rules are then translated separately. While simpler, this second option is likely to be less efficient however.

Chapter 14

Query Plans and Incremental Evaluation

So far, we have translated rules into expressions of our Complex Event Relational Algebra (CERA). While these expressions describe abstractly the operations involved in query evaluation, they do not describe how query evaluation performed in a step-wise manner with new event arriving *over time* and how it can be efficient with an *incremental evaluation* that avoids recomputing some intermediate results in each step.

As we will see, incremental evaluation depends heavily on which intermediate results we "materialize," that is, store across the different evaluation steps. To capture this information we introduce query plans that have so-called materialization points. A materialization point corresponds to a materialized intermediate result (or some output or input). The importance of materialization points goes beyond incremental evaluation, however. They help us with the evaluation of hierarchical rule programs (not just single rules), where results of one rule can be input to another through rule chaining, and can account for multi-query optimizations, where results of common subexpressions are shared. Garbage collection, which will be the topic of Chapter 15, will also refer to materialization points.

We now first explain the general ideas behind incremental evaluation of complex event queries to motivate the need for materialization points (Section 14.1). We then formally define query plans with materialization points (Section 14.2) and address their incremental evaluation by applying a technique called finite differencing (Section 14.3). We also give the translation of rule programs into such query plans (Section 14.4) and discuss how query rewriting can be applied as an optimization technique to query plans (Section 14.5), focusing especially on rewritings that go beyond the traditional rewritings of relational algebra. Finally, we discuss the relationship of the query plans and their incremental evaluation as introduced here with related work (Section 14.6).

14.1 Incremental Evaluation Explained

Recall from Chapter 13 that the event stream is represented by a relation E with schema $sch(E) = \{e.s, e.e, term\}$. A CERA expression Q that translates a complex event query rule has the similar schema $sch(Q) = \{r.s, r.e, term\}$. The occurrence time of an event tuple q of Q is $m_Q(q) := \max\{q(i.e) \mid i.e \in sch_{end}(Q_k)\}$. To express m_Q on the syntactic level of the algebra we used the shorthand $M_Q := \max\{i_1.e, \ldots, i_n.e\}$ where $\{i_1.e, \ldots, i_n.e\} = sch_{end}(Q)$.

The task of evaluating a complex event query given by a CERA expression Q is a step-wise procedure over time: A step is initiated by one or more base events (an event not defined by a rule or a timer event) happening at the current time, which we denote *now*. We assume for simplicity here that the base events are processed in the temporal order in which they happen, i.e., with ascending end time stamps. Extensions where the order of events is "scrambled" (within a known bound) are possible, however (see Chapter 18). Note that while the time domain can be

continuous (e.g., isomorphic to real numbers), the number of evaluation steps is discrete since we assume a discrete number of incoming events.

The required output for the evaluation step then is all result tuples q of Q that "happen" at the current time *now*, i.e., where $m_Q(q) = q(r.e) = now$. In other words in each step, we are not interested in the full result of Q, but only in $\triangle Q := \sigma[M_Q = now](Q)$.

The computation of $\triangle Q$ does not involve any knowledge about future events (i.e., events might happen later than the current time *now*). The temporal preservation property of CERA (cf. Chapter 13.3) ensures this and thus makes it possible to deliver results of Q immediately at their occurrence time.

However, the computation of $\triangle Q$ usually requires knowledge of past events (i.e., events that have happened *before* the current time *now*). Accordingly, in each step we also have to maintain some data structures that store knowledge about these events for use in future evaluation steps. These data structures are called event histories.

A naive, non-incremental way of query evaluation would be: Maintain a stored version of the event stream so far across steps as a relation.[1] (And similarly stored versions of the auxiliary relations for the relative timer events.) In each step simply insert all new events into the relation and evaluate the CERA expression Q with these stored relations as input from scratch according to its non-incremental semantics (see Chapter 13.2). Then apply the selection $\sigma[M_Q = now]$ to output the result of the step.

Such a naive, non-incremental query evaluation is, however, rather inefficient: we compute not only the required result $\triangle Q = \sigma[M_Q = now](Q)$, but also all results from previous steps, i.e., also $\sigma[M_Q < now](Q)$.

It is more efficient to use an incremental approach, where we store not just the input relations of Q but also some intermediate results. In each step we then compute only the changes to the result of Q and to the intermediate results. It turns out that due to the temporal preservation of CERA, the change to (the result of) Q only involves inserting new tuples, and that these tuples are exactly the ones from $\triangle Q$, which is also required as the result of the evaluation step. Similarly, the change to each intermediate result V involves only inserting new tuples, and these are $\triangle V := \sigma[M_Q = now](V)$.

This general idea leaves us with the following questions that will be addressed in the remainder of this chapter:

- Which intermediate results will be materialized (i.e., stored across steps different evaluation steps)?
- How do we compute $\triangle Q$ (as well as the $\triangle V$s) efficiently (i.e., in an incremental manner that avoids unnecessary recomputations by using the materialized intermediate results)?
- How can we extend the incremental evaluation of a single CERA expression that translates a single rule to an incremental evaluation of a hierarchical rule program (in particular one that uses rule chaining)?
- How can we perform important optimizations (e.g., multi query optimizations) on the level of the logical query plan?

14.2 Query Plans with Materialization Points

We now introduce so-called query plans with materialization points. Materialization points serve to describe which intermediate results will be materialized in the incremental evaluation (with the aim of avoiding their re-computation). Further they will account for the flow of information from one rule to another, i.e., chaining of rules, when we translate full rule programs (not just single rules).

[1]Somewhat more intelligently, we could store only the results of the matching operators $\mathsf{Q}^{\mathsf{X}}_{i:q}(E)$ in separate, individual relations instead of storing of the actual event stream in a single relation. The issues of the non-incremental evaluation that are illustrated here however remain.

The decision whether some intermediate result should be materialized or recomputed in the incremental evaluation is always a trade-off. Simply put, materializing an intermediate result should save computation time at the price of consuming more memory compared to recomputing the intermediate result. The trade-off is however not always that simple. Materializing intermediate results entails a need for garbage collection (cf. Chapter 15) and this garbage collection requires computation time that is not needed when recomputing results. The increased demand of memory can lead to the materialized results being stored at a lower level in the memory hierarchy (e.g., disc instead of main memory).[2] The time needed to access a lower level in the memory hierarchy might outweigh the benefit of a materialized result compared to recomputing it (from data at a higher level in the memory hierarchy).

Therefore in our operational semantics for XChange$^{\mathrm{EQ}}$ we do not want to commit to a single, fixed strategy for all queries that determines which intermediate results are materialized. Rather we want to develop a framework with our query plans that allows to describe each of the many possibilities so that query plans can be the output of a query compiler and optimizer that uses various heuristic and cost-based planning to compare different options.

14.2.1 Definition

The intuition of our query plans is that they describe relations that represent the materialized intermediate results (and the final results, i.e., the outputs of rules). These relations are called materialization points and the query plan contains equations that describe the contents of materialization points with CERA expressions. Evaluation steps of the incremental evaluation update the contents of materialization points with new events so that after each step the contents of a materialization point reflect the result of its defining CERA expression. As we will see in Section 14.3, the update of a materialization point is computed not with the defining CERA expression itself but more efficiently.

Definition A query plan is a sequence

$$QP = \langle \; Q_1 := E_1, \; \ldots, \; Q_n := E_n \; \rangle$$

of materialization point definitions $Q_i := E_i$. Q_i is a relation name and called a materialization point. E_i is either a CERA expression or a union $R_1 \cup \ldots \cup R_n$ of relations. The relations R_1, …, R_n can be base relations[3] or materialization points. Each materialization point Q_i is defined only once in QP, i.e., $Q_i \neq Q_j$ for all $1 \leq i < j \leq n$. The materialization point definitions must be acyclic, i.e., if Q_j occurs in E_i then $j < i$ for all $1 \leq i \leq n$ and all $1 \leq j \leq n$.

Remarks Unions $R_1 \cup \ldots \cup R_n$ are needed in query plans amongst others to properly translate some rule programs such as $A(x) \leftarrow i : B(x), j : C(x)$, $A \leftarrow i : B(x), j : D(x)$. A union could also have been added as an operator to CERA itself — temporal preservation, the central property of CERA needed for an incremental evaluation, would be unaffected. However allowing unions only at materialization points not at arbitrary places in CERA expressions is convenient for the to keep the presentation of the algorithm for determining temporal relevance in the next chapter simple. (Chapter 15.3.7 explains the necessary extensions to allow unions in arbitrary places.)

The restriction to acyclic definitions of materialization points makes their semantics straightforward (see below). For translating hierarchical rule programs our query plans are sufficient. For translating more complicated rule programs that are stratified but not hierarchical, the notion of

[2]The same principle also applies between levels in the memory hierarchy, e.g, main memory and processor caches, although there is an important difference: Between disc and main memory, the decision which data is stored where can be controlled by the query engine. Between main memory and processor cache, it is controlled by the caching strategy of the processor.

[3]When translating an XChange$^{\mathrm{EQ}}$ program, the base relations are the relation E representing the incoming event stream and the auxiliary relations X_i for relative timer events

query plans would have to be extended and a slightly more involved semantics needed. We favor simplicity of presentation here and focus on hierarchical programs. Once that is understood, the extension to stratified programs is not hard and there are no aspects to it that are specific to complex event queries.

14.2.2 Semantics

Since a query plan is acyclic, its semantics are straightforward: compute the results of its expressions from left to right (from $Q_1 := E_1$ to $Q_n := E_n$), replacing references to materialization points Q_j in expressions E_i with their (already computed) result. The result of a query plan QP is then just the sequence of the contents (i.e., relations) of the materialization points.

Note that the semantics of query plans still have the "omniscient" perspective like CERA expressions and do not (yet) reflect the step-wise evaluation over time.

14.2.3 Temporal Preservation

The temporal preservation property of CERA continues to hold for query plans. That is, to compute the contents of the materialization points in QP up to a time point *now* it suffices to know the input relations up to that time point *now*. No knowledge about the future (i.e., tuples with occurrence times after *now*) is needed. The rationale for this is that temporal preservation holds for any CERA expression as well as for unions of relations, and query plans contain nothing else.

14.2.4 Example

To illustrate query plans consider the following XChange$^{\text{EQ}}$ rule:

```
DETECT
  d [var X, var Y]
ON
  and {
    event i: a {{ var X }},
    event j: b [ var X, var Y ],
    event k: c {{ var Y }}
  }
END
```

Figure 14.1 shows two possible query plans for evaluating this rule. We can easily see that both query plans correspond to the CERA expression obtained from the translations of that rule as it has been described in the previous chapter: from top to bottom, we simply have to expand references to materialization points in expressions on the right hand sides with their definitions.

While both query plans obviously deliver the same results for Q, they materialize different intermediate results. Both query plans materialize the variable bindings obtained from matching the simple event queries in materialization points A, B, C. (It is generally a good strategy to materialize these because this way we avoid storing the incoming event stream E and avoid repeating in each step the matching of simple event queries, which is a potentially expensive operation.) Query plan 1 then uses these directly to compute the final result Q. In the incremental evaluation it will therefore compute the join $A \bowtie B$ in each step and not utilize the intermediate result of that join from the previous step. Query plan 2 "remembers" intermediate results for $A \bowtie B$ across steps and uses them in its evaluation of Q. This is indicated by introducing the materialization point V for $A \bowtie B$.

We can see from this that the difference between $(A \bowtie B) \bowtie C$ and $V \bowtie B$ with $V := A \bowtie B$ is not just an insignificant change in notation. While both query plans give the same result for Q, they are different in the incremental evaluation because they materialize different intermediate results. This will become clear in Section 14.3.

As already discussed at the beginning of this section, the performance of a query plan depends on many aspects such as characteristics of the event stream, available main memory, used physical

Query Plan 1:

$$
\begin{array}{lll}
A & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{a\{\{var X\}\}}](E) \\
B & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{b[var X, var Y]}](E) \\
C & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{c\{\{var Y\}\}}](E) \\
Q & := & \pi[r.s, r.e, term](\\
 & & \quad \gamma[r.s, r.s, i.ref, j.ref, k.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{d[var X, varY]}]](\\
 & & \quad\quad (A \bowtie B) \bowtie C))
\end{array}
$$

Query Plan 2:

$$
\begin{array}{lll}
A & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{a\{\{var X\}\}}](E) \\
B & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{b[var X, var Y]}](E) \\
C & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{c\{\{var Y\}\}}](E) \\
V & := & A \bowtie B \\
Q & := & \pi[r.s, r.e, term](\\
 & & \quad \gamma[r.s, r.s, i.ref, j.ref, k.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{d[var X, varY]}]](\\
 & & \quad\quad V \bowtie C))
\end{array}
$$

Figure 14.1: Two different example query plans for the same rule

operator implementations, etc. Therefore there is no general principle to tell which of the two query plans presented here for the example rule would be more efficient. The aim of the query plans that have been defined here is to give a representation to the different options of materializing intermediate results in a way that a query compiler and optimizer can compare different alternatives and choose.

14.3 Incremental Evaluation and Finite Differencing

We have already explained the basic idea of step-wise evaluation over time in Section 14.1 for a single CERA expression Q. The idea extends straightforwardly to query plans QP. In the incremental evaluation, we are not only concerned with computing the output in each step, but also with maintaining the event histories (i.e., the contents of the materialization points). We will see, however, that solving one issue also solves the other.

14.3.1 Input and Output in Incremental Evaluation

Each step of the incremental evaluation of a query plan $QP = \langle Q_1 := E_1, \dots, Q_n := E_n \rangle$ is initiated by one or more base events happening at the current time, which we denote *now*. The available **input** of the evaluation step is:

- $\triangle B_1, \dots, \triangle B_m$ are relations that contain the new base events that happen at the current time *now*. We have $\triangle B_i = \sigma[M_{B_i} = now](B_i)$, where B_i is the conceptual "omniscient" base relation that contains all events that ever happen (past, present, and future).
- $\circ Q_1, \dots, \circ Q_n$ are relations for the event histories of the materialization points that store results and intermediate results from previous evaluation steps. We have $\circ Q_i = \sigma[M_{Q_i} < now](Q_i)$, where Q_i is the "omniscient" result for Q_i (as described in Section 14.2.2).
- $\circ B_1, \dots, \circ B_m$ are relations for the event histories of the base events. We have $\circ B_i = \sigma[M_{B_i} < now](B_i)$. In practice, $\circ B_1, \dots, \circ B_n$ are often not needed because we have materialization points that capture their information.[4]

[4] An example of this are the two query plans from Section 14.2.4. For their incremental evaluation the history $\circ E$ of the stream of incoming events E is not needed because the histories $\circ A$, $\circ B$, $\circ C$ of the materialization points A, B, and C store all information that is needed in the incremental evaluation.

$$
\begin{array}{lcl@{\qquad}lcl}
\triangle\sigma_C(E) & = & \sigma_C(\triangle E) & \circ\sigma_C(E) & = & \sigma_C(\circ E) \\
\triangle\rho_A(E) & = & \rho_A(\triangle E) & \circ\rho_A(E) & = & \rho_A(\circ E) \\
\triangle\pi_P(E) & = & \pi_P(\triangle E) & \circ\pi_P(E) & = & \pi_P(\circ E) \\
\triangle\mu_M(E) & = & \mu_M(\triangle E) & \circ\mu_M(E) & = & \mu_M(\circ E) \\
\triangle\gamma_G(E) & = & \gamma_G(\triangle E) & \circ\gamma_G(E) & = & \gamma_G(\circ E) \\
\triangle\mathsf{Q}^{\mathsf{X}}_{i:q}(E) & = & \mathsf{Q}^{\mathsf{X}}_{i:q}(\triangle E) & \circ\mathsf{Q}^{\mathsf{X}}_{i:q}(E) & = & \mathsf{Q}^{\mathsf{X}}_{i:q}(\circ E) \\
\triangle(E_1 \cup E_2) & = & \triangle E_1 \cup \triangle E_2 & \circ(E_1 \cup E_2) & = & \circ E_1 \cup \circ E_2 \\
\triangle(E_1 \bowtie E_2) & = & \triangle E_1 \bowtie \circ E_2 \cup & \circ(E_1 \bowtie E_2) & = & \circ E_1 \bowtie \circ E_2 \\
 & & \triangle E_1 \bowtie \triangle E_2 \cup & & & \\
 & & \circ E_1 \bowtie \triangle E_2 & & & \\
\triangle(E_1 \bowtie_{i \sqsupseteq j} E_2) & = & \triangle E_1 \bowtie_{i \sqsupseteq j} \circ E_2 \cup & \circ(E_1 \bowtie_{i \sqsupseteq j} E_2) & = & \circ E_1 \bowtie_{i \sqsupseteq j} \circ E_2 \\
 & & \triangle E_1 \bowtie_{i \sqsupseteq j} \triangle E_2 & & & \\
\triangle(E_1 \overline{\ltimes}_{i \sqsupseteq j} E_2) & = & \triangle E_1 \overline{\ltimes}_{i \sqsupseteq j} \circ E_2 \cup & \circ(E_1 \overline{\ltimes}_{i \sqsupseteq j} E_2) & = & \circ E_1 \overline{\ltimes}_{i \sqsupseteq j} \circ E_2 \\
 & & \triangle E_1 \overline{\ltimes}_{i \sqsupseteq j} \triangle E_2 & & &
\end{array}
$$

Figure 14.2: Equations for finite differencing

The **output** of the evaluation step must be:

- $\triangle Q_1, \ldots \triangle Q_n$ the results for current step for all materialization points. As explained earlier, they must be $\triangle Q_i = \sigma[M_{Q_i} = now]$.

In addition to producing the output, the evaluation step must perform an important **side-effect** as preparation for future evaluation steps:

- After the evaluation step, the event histories $\circ B_1, \ldots, \circ B_m, \circ Q_1, \ldots, \circ Q_n$ must be updated (to $\circ B'_1, \ldots, \circ B'_m, \circ Q'_1, \ldots, \circ Q'_n$) so that they can become the input of the next evaluation step. This means that we must have $\circ B'_i = \sigma[M_{B'_i} \leq now](B_i)$ and $\circ Q'_i = \sigma[M_{Q'_i} \leq now](Q_i)$.

Since $\circ Q'_i = \circ Q_i \cup \triangle Q_i$ (analogous for $\circ B'_i$), computing the output also solves the main issue of the side-effect. Therefore, the main concern of the incremental evaluation is to compute the $\triangle Q_i$'s efficiently.

14.3.2 Finite Differencing

We can compute each $\triangle Q_i$ for a materialization point $Q_i := E_i$ in QP efficiently using the changes $\triangle R_j$ to its input relations R_j, together with $\circ R_j = \sigma[M_{R_j} < now](R_j)$, their materialized histories from the previous evaluation step. Note that the input relations R_j can be base relations ($B_1, \ldots, B_n$) or other materialization points that are defined (and computed) earlier in the query plan ($Q_1, \ldots, Q_{i-1}$).

Using a technique called finite differencing, we can derive a relational algebra expression $\triangle E_i$ so that $\triangle E_i$ involves only $\triangle R_j$ and $\circ R_j$ and $\triangle E_i = \triangle Q_i$ (for each step). Finite differencing works by pushing the differencing operator $\triangle$ inwards according to the equations in Figure 14.2. The equations might yield expressions where the "history operator" $\circ$ is applied to an expression that is not a base relation or materialization point. In those cases, we also need to push the "history operator" $\circ$ inwards; the appropriate equations are also given in Figure 14.2.

Finite differencing is a method originating in the incremental maintenance of materialized views in databases, which is a problem very similar to incremental event query evaluation [GL95]: the materialization points in our query plans can be understood as definitions of (materialized) views that must be updated ("maintained") whenever new events are added to their input relations. In contrast to the general view maintenance problem, however, we only have to consider adding events (not also removing or changing them). This is due to the temporal preservation of CERA. An extension where we would also consider removing events (by introducing $\triangledown Q_i$ and $\triangledown R_i$) would be interesting to deal with out of order arrival of events (cf. Chapter 18)

Note that finite differencing has similarities with obtaining the derivate of a function through symbolic differentiation (e.g., with equations such as $\frac{d}{dx}(fg) = f\frac{d}{dx}g + \frac{d}{dx}fg$) in mathematical calculus. However, finite differencing is not concerned with a differential quotient $\frac{f(x+\triangle x)-f(x)}{\triangle x}$ but the finite difference of the contents of an event history between two different steps. If we see an event history $\circ Q$ as a time-varying function $f(t)$, then this difference is the set difference $f(t+\triangle t) - f(t)$ where $\triangle t$ is the time that elapses between two steps.

14.3.3 Correctness

To show that the equations for finite differencing of Figure 14.2 are correct, we have to show that for all time points *now* and all materialization points $Q := E$ (E a CERA expression or a union of relations) it holds that

$$\triangle E = \sigma[M_Q = now](Q) \quad \text{and} \quad \circ\, E = \sigma[M_Q < now](Q)$$

provided that

$$\triangle R_i = \sigma[M_{R_i} = now](R_i) \quad \text{and} \quad \circ\, R_i = \sigma[M_{R_i} < now](R_i)$$

for all the input relations R_i of E.

The proof is a simple structural induction on E and makes similar arguments about time stamps (and related restrictions of CERA compared to traditional relational algebra) as the proof of the temporal preservation property of CERA (cf. Chapter 13.3 and Appendix B.1).

Note that finite differencing of arbitrary relational algebra expressions is not always as simple as it is here for CERA. In traditional algebra, care must be taken for example with projections that $\triangle\pi_P(E)$ does not produce any "duplicate" tuples that are already in $\circ\pi_P(E)$ (and therefore the equation $\triangle\pi_P(E) = \pi_P(\triangle E)$ does not hold in general for relational algebra). Further new tuples on the right hand side of an anti-semi-join or a difference might actually remove tuples from the result so that there not tuples that are added to the result ($\triangle E$) play a role but also tuples that are removed from the result ($\triangledown E$). The time stamps that are part of every relation and the related restrictions in CERA make finite differencing much easier because they enure that there are no difficulties with respect to duplicates and no tuples must ever be removed from the result.[5]

14.3.4 Finite Differencing of Multiple Joins

When applying finite differencing to expressions with multiple joins such as $E = R \bowtie S \bowtie T$ (or more generally $E = R_1 \bowtie \ldots \bowtie R_n$), the equations of Figure 14.2 have a disadvantage: the resulting expression $\triangle E$ is exponential in size compared to the original expression E and some subexpressions occur multiple times. For example

$$E = R \bowtie S \bowtie T$$

gives

$$\begin{array}{rclcl} \triangle E & = & \triangle R & \bowtie & (\circ S \bowtie \circ T) \\ & & \triangle R & \bowtie & (\triangle S \bowtie \circ T \;\cup\; \triangle S \bowtie \triangle T \;\cup\; \circ S \bowtie \triangle T) \\ & & \circ R & \bowtie & (\triangle S \bowtie \circ T \;\cup\; \triangle S \bowtie \triangle T \;\cup\; \circ S \bowtie \triangle T)\,. \end{array}$$

Notice that the subexpression $(\triangle S \bowtie \circ T \;\cup\; \triangle S \bowtie \triangle T \;\cup\; \circ S \bowtie \triangle T)$ occurs twice. For a join of n relations $E = R_1 \bowtie \ldots \bowtie R_n$, the resulting expression $\triangle E$ will have a size of $O(2^n)$ compared to the original expression E. (Each of the $n-1$ joins doubles the size of the original expression.)

[5] Note that tuples being removed from the result would be a big difficulty for the step-wise evaluation of event queries over time: in essence it would mean that the event query evaluation gives an answer at one point in time and "retracts" it at a later point in time.

An alternative equation for the finite differencing of a join of n relations that would avoid that the same subexpression occurs several times would be:

$$\begin{array}{rcllllllll}
\triangle(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) & = & \triangle R_1 & \bowtie & \triangle R_2 & \bowtie & \ldots & \bowtie & \triangle R_n & \cup \\
& & \circ R_1 & \bowtie & \triangle R_2 & \bowtie & \ldots & \bowtie & \triangle R_n & \cup \\
& & \triangle R_1 & \bowtie & \circ R_2 & \bowtie & \ldots & \bowtie & \triangle R_n & \cup \\
& & \circ R_1 & \bowtie & \circ R_2 & \bowtie & \ldots & \bowtie & \triangle R_n & \cup \\
& & \ldots & & & & & & & \cup \\
& & \triangle R_1 & \bowtie & \triangle R_2 & \bowtie & \ldots & \bowtie & \circ R_n & \cup \\
& & \circ R_1 & \bowtie & \triangle R_2 & \bowtie & \ldots & \bowtie & \circ R_n & \cup \\
& & \triangle R_1 & \bowtie & \circ R_2 & \bowtie & \ldots & \bowtie & \circ R_n &
\end{array}$$

This expression basically makes a union of all combinations of $\triangle R_i$ and $\circ R_i$, except for $\circ R_1 \bowtie \circ R_2 \bowtie \ldots \bowtie \circ R_n$. In total there are $2^n - 1 = O(2^n)$ such combinations, so the length of resulting expression is still exponential.

However, this equation is very systematic so that there is no need to explicitly represent it in an implementation of the query evaluation engine. (It can internally just use the original expression E instead of $\triangle E$). This is particularly interesting because in practice in each step most $\triangle R_i$'s will be empty anyway and only one or two $\triangle R_i$ contain new event tuples. The joins in the union containing at least one $\triangle R_i = \emptyset$ will deliver an empty result, so that it suffices to consider only those few joins that contain at least one $\triangle R_i \neq \emptyset$.

A further alternative would be the following equation for the finite differencing of a join of n relations.[6]

$$\begin{array}{ll}
\multicolumn{2}{l}{\triangle(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)} \\
= & \triangle R_1 \bowtie \circ R_2 \bowtie \ldots \bowtie \circ R_n \cup \\
& \triangle(R_2 \bowtie R_n) \bowtie (\triangle R_1 \cup \circ R_1) \\
= & \triangle R_1 \bowtie \circ R_2 \bowtie \ldots \bowtie \circ R_n \cup \\
& \triangle R_2 \bowtie \circ R_3 \bowtie \ldots \bowtie \circ R_n \bowtie (\triangle R_1 \cup \circ R_1) \cup \\
& \ldots \cup \\
& \triangle R_i \bowtie \circ R_{i+1} \bowtie \ldots \bowtie \circ R_n \bowtie (\triangle R_1 \cup \circ R_1) \bowtie \ldots \bowtie (\triangle R_{i-1} \cup \circ R_{i-1}) \cup \\
& \ldots \cup \\
& \triangle R_n \bowtie (\triangle R_1 \cup \circ R_1) \bowtie \ldots \bowtie (\triangle R_{n-1} \cup \circ R_{n-1})
\end{array}$$

The length of the resulting expression is quadratic in the size of the original expression. Particularly interesting about expressions that take this form is that they contain subexpressions of the form $\triangle R_i \cup \circ R_i$, which is turn is just $\circ R_i'$ (cf. Section 14.3.1), the value that $\circ R_i$ should have after each evaluation step. When reading (and evaluating) the subexpressions of the union from top to bottom, then all subexpressions before the ith (i.e., before the line starting with $\triangle R_i \bowtie \ldots$) access only $\circ R_i$ and all subexpressions after it only $\triangle R_i \cup \circ R_i$. As long as $\circ R_i$ is not accessed in any other materialization points of a query plan, the side-effect $\circ R_i' = \triangle R_i \cup \circ R_i$ can be performed immediately when evaluating the ith subexpression.[7]

14.3.5 Overall Query Evaluation Algorithm

To get back to the overall picture of incremental event query evaluation, let us again consider how a given query plan $QP = \langle Q_1 := E_1, \ldots, Q_n := E_n \rangle$ is conceptually evaluated. Let QP use the base relations $B_1, \ldots, B_m$.

[6]Note that this equation has different join orders in the different subexpressions of the union. Some care is therefore necessary when trying to transfer this equation from the named perspective on relational algebra to the unnamed, positional (which is relevant for an implementation). Under the named perspective $R \bowtie S = S \bowtie R$. Under the unnamed perspective, however, $S \bowtie R$ has a different order of attributes than $R \bowtie S$. This different order must be "rectified" with a projection (which might be implemented as part of the join operation).

[7]This is particularly advantageous for hash joins: we only have to compute the hash value for a tuple $r \in \triangle R_i$ once and use it both for joining r with $\circ R_{i+1}, \ldots, \triangle R_{i-1} \cup \circ R_{i-1}$ and for inserting r into $\circ R_i'$.

As part of query compilation, we apply finite differencing to QP by simply applying it to every materialization point definition $Q_i := E_i$. The result will be written $\triangle QP = \langle \triangle Q_1 := \triangle E_1, \ldots \triangle Q_n := \triangle E_n \rangle$. Note that the base relations of $\triangle QP$ are $\triangle B_1, \ldots \triangle B_m, \ldots \circ B_1, \ldots \circ B_m, \circ Q_1, \ldots \circ Q_n$.

The evaluation of QP then conceptually follows the following schema, where each iteration of the (infinite) loop corresponds to an evaluation step.

$\circ B_1 := \emptyset; \ldots; \circ B_m := \emptyset;$
$\circ Q_1 := \emptyset; \ldots; \circ Q_n := \emptyset;$

`while(true)` {
  advance *now* to the occurrence time of the next incoming event(s);
  $\triangle Q_1 := \emptyset; \ldots; \triangle Q_n := \emptyset;$
  initialize $\triangle B_1, \ldots, \triangle B_m$ with the current events;
  compute $\triangle Q_1, \ldots, \triangle Q_n$ according to $\triangle QP$;
  `for` $i := 1 \ldots n$ {
    $\circ Q_i := \circ Q_i \cup \triangle Q_i;$
  }
  output $\triangle Q_1, \ldots, \triangle Q_n;$
}

The evaluation of $\triangle QP$ in each step is as described in Section 14.2.2. Keep in mind however that the base relations are $\triangle QP$ are $\triangle B_1, \ldots \triangle B_m, \ldots \circ B_1, \ldots \circ B_m, \circ Q_1, \ldots \circ Q_n$ (not $B_1, \ldots, B_m$ as in the original, "omniscient" QP). Also, expressions of materialization point definitions in $\triangle QP$ can contain unions at arbitrary places $\triangle QP$.

14.4 Translation of Rule Programs into Query Plans

Having formally defined the notion of query plans and considered how they are evaluated incrementally, we now turn to translating a given XChange$^{\mathrm{EQ}}$ program P into a query plan QP. This QP is just a first query plan for P that can then be optimized further (e.g., through rewritings that are discussed in Section 14.5). The most important part of the work, translating single rules into CERA expressions, has already been addressed in the previous chapter. The main issue that we must address now is rule chaining, and here the materialization points in our query plan will help us greatly.

14.4.1 Basic Translation

Let $P = P_1 \uplus \ldots \uplus P_n$ be a stratification of the hierarchical XChange$^{\mathrm{EQ}}$ program P. It is preferable but not necessary that there are as few strata as possible. The query plan QP will be the concatenation of n sequences of materialization point definitions, where each such sequence corresponds to one stratum. The idea is that each rule r gives rise to one materialization point definition Q using the translation from the previous chapter. Additional materialization points Z "collect" the results of all rules of stratum so that the rules of the next higher stratum use Z as their incoming event stream. This additional materialization point therefore solves the issue of rule chaining. (Improvements on this first translation will be discussed later.)

Let $P_1 = \{r_1, \ldots, r_k\}$ be the rules of the first stratum. We translate it into a sequence

$$
\begin{array}{llll}
S_1 = \langle & Q_{1,1} & := & \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\mathrm{translate}_E(r_1)) \\
& \ldots, & & \\
& Q_{1,k} & := & \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\mathrm{translate}_E(r_k)), \\
& Z_1 & := & E \cup Q_{1,1} \cup \ldots \cup Q_{1,k} \qquad \rangle
\end{array}
$$

of materialization points definitions. Here, $\mathrm{translate}_E(r_i)$ denotes the translation of the rule r_i

into a CERA expression with the incoming event stream being E. We deviate in one point from the way the translation has been described in the previous chapter:

For higher strata $P_i = \{r_1, \ldots, r_l\}$, the sequence is similar, only the translation of rules uses Z_{i-1} instead of E the as incoming event stream, so that the "input" to rules in stratum P_i includes events that have been derived in lower strata.

$$
\begin{array}{llll}
S_i = \langle & Q_{i,1} & := & \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\text{translate}_{Z_{i-1}}(r_1)), \\
& \ldots, & & \\
& Q_{i,l} & := & \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\text{translate}_{Z_{i-1}}(r_l)), \\
& Z_i & := & Z_{i-1} \cup Q_{i,1} \cup \ldots \cup Q_{i,l} \qquad \rangle
\end{array}
$$

The query plan QP for P then is just the concatenation of $S_1, \ldots, S_n$: $QP = S_1 \cdot \ldots \cdot S_n$. Because the program P was hierarchical, QP is acyclic as required.

14.4.2 Correctness

The intended output of QP is Z_n; for a correct query plan, it must hold that $Z_n = M_{P,E}$, where $M_{P,E}$ is the fixpoint interpretation of P under E as defined in Chapter 10. We sketch the main idea of the proof here shortly.

By induction on i we show that each Z_i in QP computes exactly $M_i = T^{\omega}_{\overline{P_i}}(M_{i-1})$ (with $M_0 = E$). Because our program is hierarchical, we have $T^{\omega}_{\overline{P_i}}(M_{i-1}) = T_{\overline{P_i}}(M_{i-1})$. With this, both the proofs for the basis (Z_i) and induction step ($Z_{i-1} \to Z_i$) just rely on the correctness of the translation of an individual rule (see previous chapter).

It follows then that $Z_n = M_n = M_{P,E}$.

14.4.3 Example

As an example for the translation of a rule program into a query plan consider following program consisting of three rules shown in Figure 14.3(a). The stratification $P = P_1 \uplus P_2$ that we use for the translation has two strata, P_1 contains the first two rules, P_2 the third rule. The corresponding query plan in shown in Figure 14.3(b).

14.4.4 Improvements

There are two important improvements that can be made to the translation of a rule program into an initial query plan as it has just been described. All these improvements could also be made by means of rewriting the query plan as will be discussed in Section 14.5. However, the improvements that we discuss here are of a nature that one might decide to always perform them without the need for comparing and exploring alternatives in a branch-and-bound style of a query planner. In this case, applying the improvements as part of the translation phase is generally easier and with less computational overhead than in the query rewriting phase.

Materialization points for simple event queries In the example of Section 14.2.4, we have discussed that it is generally a good strategy to have materialization points for all simple event queries. This avoids repeating the potentially expensive pattern-matching operation in each evaluation step (note that this operation would have to be preformed not just for the incoming events of the current step but all events in the history of the event stream). If we have a materialization point for every simple event query, we further can avoid maintaining a history of the event stream (and also histories for the materialization points Z_i).[8]

When we apply this improvement to the example of Figure 14.3(b), it leads to the improved query plan in Figure 14.4. Note that here we have recognized that the simple event query `c[var X]` is shared in Q_1 and Q_2 and thus only introduce one materialization point for it. (In Q_2 then

[8]The algorithm for determining temporal relevance of the next chapter will detect whether a history for the incoming event stream and any materialization point is needed or not.

```
DETECT
  y[u[var X]]
ON
  and {
    event i: a[var X],
    event j: b[var X],
    event k: c[var X]
  }
END
```

```
DETECT
  y[v[var X]]
ON
  and {
    event i: c[var X],
    event j: d[var X]
  }
  where { i before j }
END
```

```
DETECT
  z[var X]
ON
  and {
    event i: f[var X],
    event j: y[var X]
  }
  where { {i,j} within 4h }
END
```

(a) Example Rule Program

$$
\begin{aligned}
QP = \langle\ Q_1 &:= \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
&\pi[r.s, r.e, term](\\
&\gamma[r.s, r.e, i.ref, j.ref, k.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{y[u[var X]]}]](\\
&\mu[r \leftarrow i \sqcup j \sqcup k](\\
&\quad \mathsf{Q}^{\mathsf{X}}[i : \texttt{a[var X]}](E) \bowtie\\
&\quad \mathsf{Q}^{\mathsf{X}}[j : \texttt{b[var X]}](E) \bowtie\\
&\quad \mathsf{Q}^{\mathsf{X}}[k : \texttt{c[var X]}](E)\quad)))),\\
Q_2 &:= \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
&\pi[r.s, r.e, term](\\
&\gamma[r.s, r.e, i.ref, j.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{y[v[var X]]}]](\\
&\mu[r \leftarrow i \sqcup j](\\
&\sigma[i.e < i.s](\\
&\quad \mathsf{Q}^{\mathsf{X}}[i : \texttt{c[var X]}](E) \bowtie\\
&\quad \mathsf{Q}^{\mathsf{X}}[j : \texttt{d[var X]}](E)\quad))))),\\
Z_1 &:= E \cup Q_1 \cup Q_2,\\
Q_3 &:= \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
&\pi[r.s, r.e, term](\\
&\gamma[r.s, r.e, i.ref, j.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{z[var X]}]](\\
&\mu[r \leftarrow i \sqcup j](\\
&\sigma[\max\{i.e, j.e\} - \min\{i.s, i.s\} \leq 4](\\
&\quad \mathsf{Q}^{\mathsf{X}}[i : \texttt{f[var X]}](Z_1) \bowtie\\
&\quad \mathsf{Q}^{\mathsf{X}}[j : \texttt{y[var X]}](Z_1)\quad))))),\\
Z_2 &:= Z_1 \cup Q_3\\
\rangle
\end{aligned}
$$

(b) Corresponding Query Plan

Figure 14.3: Example for translation of a hierarchical rule program into a query plan

$$
\begin{array}{llll}
QP & = & \langle & A := \mathsf{Q}^{\mathsf{X}}[i : \texttt{a[var X]}](E), \\
 & & & B := \mathsf{Q}^{\mathsf{X}}[j : \texttt{b[var X]}](E), \\
 & & & C := \mathsf{Q}^{\mathsf{X}}[k : \texttt{c[var X]}](E), \\
 & & & D := \mathsf{Q}^{\mathsf{X}}[j : \texttt{d[var X]}](E), \\
 & & & \\
 & & & Q_1 := \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
 & & & \quad \pi[r.s, r.e, term](\\
 & & & \quad \gamma[r.s, r.e, i.ref, j.ref, k.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{y[u[var X]]}]](\\
 & & & \quad \mu[r \leftarrow i \sqcup j \sqcup k](\\
 & & & \quad A \bowtie B \bowtie C \quad)))), \\
 & & & \\
 & & & Q_2 := \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
 & & & \quad \pi[r.s, r.e, term](\\
 & & & \quad \gamma[r.s, r.e, k.ref, j.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{y[v[var X]]}]](\\
 & & & \quad \mu[r \leftarrow k \sqcup j](\\
 & & & \quad \sigma[k.e < i.s](\\
 & & & \qquad C \bowtie D \quad))))), \\
 & & & \\
 & & & Z_1 := E \cup Q_1 \cup Q_2, \\
 & & & \\
 & & & F := \mathsf{Q}^{\mathsf{X}}[i : \texttt{f[var X]}](Z_1), \\
 & & & Y := \mathsf{Q}^{\mathsf{X}}[j : \texttt{y[var X]}](Z_1), \\
 & & & \\
 & & & Q_3 := \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](\\
 & & & \quad \pi[r.s, r.e, term](\\
 & & & \quad \gamma[r.s, r.e, i.ref, j.ref, k.ref, term \leftarrow \mathsf{C}^{\mathsf{X}}[\texttt{z[var X]}]](\\
 & & & \quad \mu[r \leftarrow i \sqcup j](\\
 & & & \quad \sigma[\max\{i.e, j.e\} - \min\{i.s, i.s\} \leq 4](\\
 & & & \qquad F \bowtie Y))))), \\
 & & & \\
 & & & Z_2 := Z_1 \cup Q_3 \\
 & & \rangle &
\end{array}
$$

Figure 14.4: Improvement: materialization points for all simple event queries

the event identifier i had to be replaced with k.) In a simple case like this, performing this (limited form of) multi-query optimization is fairly easy because the simple event queries are syntactically equal. The more general case where simple event queries are only semantically equivalent (like `a{{ b[var X], c[var Y] }}` and `a{{ c[var Z], b[var Y] }}` with appropriate renaming of the variables) is much harder. Depending on the expressivity of the underlying query language, deciding equivalence of simple event queries might have a very high complexity or be undecidable.

Note that such materialization points for simple event queries also allow us to define auxiliary relations for relative timer events using not the incoming event stream E but just the materialization point of the simple event queries used to define the timer (see also Chapter 13.4.2).

Restricting incoming information for simple event queries When we look at the materialization point F in the query plan of Figure 14.4, we can see that its matching operator matches against Z_1, which is defined as $E \cup Q_1 \cup Q_2$. However, neither Q_1 nor Q_2 can generate any simple event that would match `f[var X]`. The query plan could therefore further be improved by changing the definition of F to $F := \mathsf{Q}^{\mathsf{X}}[i : \texttt{f[var X]}](E)$.

The necessary information for detecting that results of Q_1 and Q_2 cannot be relevant for F is contained in the dependency graph that must computed as part of query compilation to determine a possible stratification for a program. Therefore this improvement, which restricts unnecessary flow of information in our query plan, is usually easier to perform as part of the translation phase than as part of query rewriting.

14.5 Query Plan Rewriting

An important motivation for introducing a formal representation of query plan as we have done in this chapter is that it enables us to use query plan rewriting as an optimization technique. Query rewriting is a central technique in the optimization of database queries and it can be expected to be of similar importance for event queries.

Query rewriting is usually based on rewriting rules that express a transformation of a given query plan into another, equivalent query plan. Equivalence between query plans means for our purposes that the result for Z_n is the same for all possible values of the incoming event stream E.

14.5.1 Traditional Relational Algebra Equivalences

Since the expression on the right hand side of materialization point definitions are based on CERA, which in turn is a variant of relational algebra, many well-known rewriting rules using laws about equivalences in relational algebra are applicable. This includes for example the following laws that give rise to corresponding rewriting rules:

- Changing join order: $R \bowtie S = S \bowtie R$, $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- Pushing selections: $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$, $\sigma_C(R \,\overline{\ltimes}_{i \sqsupseteq j}\, S) = \sigma_C(R) \,\overline{\ltimes}_{i \sqsupseteq j}\, S$, (provided that C contains only attributes from $sch(R)$)
- Changing selection order: $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_1 \wedge C_2} = \sigma_{C_2}(\sigma_{C_1}(R))$.
- Pushing projection: $\pi_P(R \bowtie S) = \pi_P(\pi_{P_1}(R) \bowtie \pi_{P_2}(S))$, where $P_1 = P \cap sch(R)$, $P_2 = P \cap sch(S)$
- Projection before grouping $\gamma_{G,a \leftarrow F(A)}(R) = \gamma_{G,a \leftarrow F(A)}(\pi_{G \cup A}(R))$ (the purpose of this law is usually to then push the projection $\pi_{G \cup A}$ further down in R with the equivalence above)

Comprehensive lists of these laws can be found in most books on databases (see, e.g., [AHV95] or [GUW01]); because they are not specific to event queries and event query plans we do not go into further detail on these rewritings.

14.5.2 Equivalences Based on Temporal Reasoning

More interesting in our context are rewritings that are specific to event queries, e.g., because the leverage temporal reasoning to simplify temporal conditions. Examples are the following equivalences:

- Simplifying maxima on time stamps: $\sigma[\max\{i_1, i_2 \dots i_n\} - \min\{j_1, \dots j_m\} \leq d \quad \wedge \quad i_1 \leq i_k](R) = \sigma[\max\{i_2 \dots i_n\} - \min\{j_1, \dots j_m\} \leq d \quad \wedge \quad i_1 \leq i_k](R)$, where $k \neq 1$; similar equivalences can be given for minima.

- Elimination of implied conditions: $\sigma[i \leq j, j \leq k, i \leq k](R) = \sigma[i \leq j, j \leq k](R)$; note that such an elimination may also use implicit assumptions about time stamps, e.g., $\sigma[i.e \leq j.s, i.s \leq j.s](R) = \sigma[i.e \leq j.s](R)$ because we always assume $i.s \leq i.e$.

14.5.3 Introduction of New Materialization Points

Even more interesting are rewritings that affect not just the right hand side of a single materialization points definition but affect the query plan as a whole. Because they may affect incremental evaluation by changing which intermediate results are materialized, these rewritings are a very important part of query optimization for event query plans.

The most important rewriting is to create a new materialization point V for some subexpression E' in a materialization point definition $Q := E$. Written as a rule, the rewriting is:

$$\frac{\begin{array}{lll} \langle & \dots, & \\ & Q := E & (E \text{ contains subexpression } E'), \\ & \dots \quad \rangle & \end{array}}{\begin{array}{lll} \langle & \dots, & \\ & V := E', & (V \text{new name}) \\ & Q := E[E'/V], & (E[E'/V] \text{ denotes replacing } E' \text{ with } V \text{ in } E) \\ & \dots \quad \rangle & \end{array}}$$

Typically this rewriting rule should only be considered when E contains at least one join inside E' and one outside E'. The reason for this is that the additional materialization point has a significant effect on incremental evaluation only when this is the case. For example when $\langle Q := R \bowtie S \bowtie T \rangle$ is changed to $\langle V := R \bowtie S,\ Q := V \bowtie T \rangle$, then the incremental evaluation will utilize stored intermediate results for $R \bowtie S$ in $\circ V$ and avoid recomputing them.

On the other hand, changing for example $\langle Q := \pi(\sigma(R)) \rangle$ to $\langle V := \sigma(R),\ Q := \pi(V) \rangle$ has little effect. The incremental evaluation of $Q := \pi(V)$ uses only $\triangle V$ and not $\circ V$, so that no benefit in terms of avoiding to recompute intermediate results in different evaluation steps is given.

Note that the rewriting rule given here could also be applied in the other direction to remove an existing materialization point (provided that V is not used in the definition of another materialization except Q). This direction is less relevant here, because we have translated rules in way that the do not create any "unwanted" materialization points (see Section 14.4). However another strategy might be to work from the opposite direction and create, e.g., a materialization point for every binary join in the translation phase and then remove "unwanted" ones in the rewriting phase. ("Unwanted" here means that heuristics or cost-measures in a query planner indicate that the query plan without that specific materialization point is more efficient.)

14.5.4 Multi-Query Optimization

A salient feature of our query plans is that they can describe multi-query optimizations well. For example the following rule will utilize the result of another materialization point V if it is equivalent to (i.e., provides the same results as) the subexpression E'' in $Q := E'$.

$$\frac{\begin{array}{ll}\langle\ \ \ldots, & \\ \quad V := E, & \\ \quad \ldots, & \\ \quad Q := E', & (E' \text{ contains subexpression } E'' \text{ with } E'' \equiv E) \\ \quad \ldots\ \ \rangle & \end{array}}{\begin{array}{l}\langle\ \ \ldots, \\ \quad V := E, \\ \quad \ldots, \\ \quad Q := E[E'/V], \\ \quad \ldots\ \ \rangle\end{array}}$$

Note that the hard problem in multi-query optimization is recognizing equivalent subexpression, i.e., that $E'' \equiv E$. It usually has a high computational complexity or might even be undecidable.[9] Our notion of query plans helps us to describe the multi-query optimization in the operational semantics but not much in recognizing possibilities for multi-query optimization.

Because event query evaluation usually entails evaluating several, often very many, event queries at the same time, multi-query optimization is of high importance there. In particular, it is more important than in databases where the traditional model is to evaluate a single query at a time. (Accordingly, multi-query optimization there is often limited to equivalent subexpressions within the same query — this can be expected to be far less the case than equivalent subexpressions over many different queries.)

14.5.5 Outlook: Query Planning in Complex Event Processing

Rewriting rules as they have been shown in this section are only one part of a query optimizer. The second part is to have good cost measures to compare alterative query plans that have been generated using the rewriting rules. Such cost measures and how to efficiently explore the search space of alternative query plans in a branch and bound manner are issues that have been investigated deeply for traditional database systems (see, e.g., [GM93, Gra95]).

The general approach of database systems transfers to event queries. However, there are two important difficulties:

- Event queries require different cost measures because they are evaluated differently. Their evaluation is usually main-memory-based whereas traditional cost measure estimate number of page accesses on disc. The goal of optimization is also different. Databases aim at reducing the overall cost, event queries often aim also at having the cost distributed well over different evaluation steps in the incremental evaluation.

- Cost measures require statistics and estimations about data distribution etc. in the input data; in the case of event queries this would mean information about the incoming event stream. Such statistics and estimations might not be available at the time of query compilation, simply because the event stream that will be received in the future is not known. In a database, all data —and thus necessary statistics and estimates— are readily available during query compilation.

One possible solution to these issues might be to generate several query plans solely based on simple heuristics that do not use cost measures, start the evaluation of all plans in parallel, and drop plans that turn out (by using appropriate measurements) to be inefficient at runtime.

[9] Note that since it is an optimization technique, it usually is not necessary to recognize all equivalences. We would be content just with recognizing many common cases. In so far, sound but incomplete recognition of equivalent subexpressions can be interesting.

14.6 Relationship with other Approaches

To conclude this chapter, we compare our approach to query plans and their incremental evaluation with other approaches to evaluate complex event queries and with approaches to other related issues.

14.6.1 RETE and TREAT

Our query plans with materialization points and their incremental evaluation can be understood as a generalization of RETE networks [For82] in the following sense: RETE always materializes the (intermediate) results of binary joins, while our query plans can choose their materialization points more freely, e.g., also materialize joins of higher arity. In this respect it is therefore also a generalization of TREAT [Mir87], which in contrast to RETE does not materialize any intermediate join results.

However our approach is also very much a restriction of RETE and TREAT in that in our incremental evaluation we only add tuples to materialization points. Deletion due to the retraction of facts, which is necessary in production rule systems, is not needed for our event query evaluation. We only need deletion as a garbage collection mechanism that does not influence query semantics (see next chapter). Neither RETE nor TREAT have any notion of automatic garbage collection in their original versions.

Our event query evaluation can process multiple tuples in a $\triangle Q$ at once, which usually is more efficient. RETE and TREAT in contrast process one tuple (also called token there) at a time. Because of the conflict resolution and side-effects of rules that are fired, extending RETE and TREAT to process multiple tuples at once would probably be hard.

Query optimization by changing the structure of a RETE network has not been explored much (usually RETE uses the join order given by the textual order in which literals occur in the production rules). In contrast, query plan rewriting has been highlighted here as an important optimization technique (although not yet explored in its full depth).

14.6.2 Event Trees and Graphs

A popular approach for evaluating event queries in composition-operator-based languages (cf. Chapter 3.2) are event trees (or graphs) [CKAK94, MSS97, ME01, AC06]. The event tree mirrors the syntactic structure of an event query expression, with each composition operator (e.g., conjunction, sequence) giving rise to a tree node. Leaves correspond to simple event queries. Simple and "semi-composed" events flow upward along the edges from the bottom to the top of the tree. Semi-composed events are sets of events that satisfy a subexpression in the operator tree; they correspond to intermediate results in CERA (e.g., the output of a join). Operators must maintain histories for some of their incoming edges. For example, a binary conjunction must have a history for both its inputs, a binary sequence only for its left input. By exploiting subtrees that are shared between several such event trees, one can build up an event graph.

The operation each node performs (i.e., producing output, storing input in the histories) is usually described only in a procedural manner and often the aspect of data in events in completely ignored. When considering the aspect of event data, many nodes have direct correspondences to relational algebra (or CERA) expressions. For example, the conjunction corresponds to a join, the sequence to a join followed by a selection expressing that the events from the left must happen before the events from the right. In as far, our query plans can be understood as a generalization of event trees that explains better how data is treated and gives a strong theoretical foundation due to its rooting in relational algebra.

Like RETE, event trees have a fixed notion which intermediate results are materialized. Transformations of event trees to make their evaluation more efficient are also difficult. Our query plans with materialization points are more flexible in both points.

14.6.3 Petri Nets and Finite State Automata

Petri nets have been suggested in [GD93, GD94] as an alternative to event trees for evaluating a composition-operator-based language. Tokens correspond to simple and semi-composed events. A close inspection of the approach shows however that they are not as different from the event tree approach as it might seem at first glance. Essentially each composition operator gives rise to one petri-net that has several incoming transitions and one outgoing transition. The petri-nets of the individual operators are then "wired" according to the syntactic structure of the event query expression. This of course means that the structure is the same as for an event tree. The only difference is that the internal implementation of the nodes is described more detailed trough its petri net. Therefore, the discussion about the differences between our query plans and event trees applies to petri nets as well.

Another approach for the evaluation of event queries coming from composition-operator-based languages are finite state automata [GJS92a, BC06]. They are based on the idea of interpreting the event stream as a sequence of "letters" and the evaluation of an event query as a string search (or more generally search for a textual pattern) in the text represented by the event stream.

Automata-based approaches are therefore particularly interesting for event queries that involve mainly detecting particular sequences of events occurring at time points. However, the approach does not explain well how to deal with events occurring over time intervals and how to deal with event data. Further note that conceptually each incoming event must start a new instance of the automaton because the event sequence to be detected can start with any event. (There are of course well-known solutions to this issue such as the Knuth-Morris-Pratt algorithm for string search, see, e.g., [KJP77]). Query optimizations (e.g., by changing the structure of the automaton) have also not been explored for automata in the context of complex event queries.

14.6.4 Query Evaluation in Data Stream Management Systems

Query evaluation in data stream management systems uses data flow networks with stateful operators that correspond the relational algebra operations [ACÇ+03, ABB+03, ABW06]. In this respect they are similar to RETE, event trees, and to our query plans. Note that most work in data stream management systems research is concerned with the lower, physical level of implementation of these operations. Little work has been done with regards to the representation and rewriting of logical query plans as it has been discussed in this chapter. The research done on data stream management systems is highly relevant for the efficient physical implementation of our logical query plans.

Like RETE and event trees, data stream management systems usually use a fixed strategy to decide which intermediate results are materialized. Usually each operator is responsible for maintaining the "event history" of its input (which is usually called a synopsis). Therefore data stream management systems also do not have the flexibility of different materialization strategies.

Most data stream management systems have a notion of "negative tuples," which are similar to deletions of tuples in RETE. (Usually these negative tuples are generated for events that leave the window that is used to turn the event stream into a relation, cf. Chapter 3.3.2.) By design of CERA and its temporal preservation property (cf. Chapter 13.3), our query plans require only propagation of positive information (finite differencing only describes new tuples $\triangle R$). As discussed earlier in Section 14.3 our approach to finite differencing could be extended to propagate also negative information (i.e., extended to also compute tuples to be removed $\triangledown R$). However, for the translation of XChange$^{\text{EQ}}$ programs, CERA is sufficient and thus also our query plans.

14.6.5 Conclusion

The logical query plans that have been presented in this chapter offer a theoretically well-founded description of the incremental evaluation of event query programs. Unsurprisingly, they describe well the operations that are performed in many related approaches to event query evaluation (in particular, RETE, TREAT, event trees, and query evaluation on data stream). A considerable

strong point of our query plans is that they can describe different strategies to materializing intermediate results (e.g., both the RETE and the TREAT strategies and any in-between solutions). Since related approaches always use a single, fixed strategy for materializing intermediate results, these approaches do not have a notion of "materialization points" like our query plans and do not separate the query plan and its incremental evaluation (through finite differencing) as we have done here. The comparison in this section also shows that many approaches to evaluating complex event queries are, at least on the logical level describing which computations must be performed, more similar than they might seem at the first glance.

Chapter 15

Relevance of Events

Evaluation of complex event queries over time involves storing information about past events (as well as intermediate results) in the event histories, as we have seen in the previous chapter. The incremental evaluation described in Chapter 14.3 simply stores all event tuples for each base relation and materialization point for an unlimited amount of time.

While it is necessary to store received events for some time, it is however often not necessary to store them forever. Temporal conditions in queries render certain events irrelevant after some time. We call the period of time for which an event or an intermediate result must (at least) be stored its *temporal relevance*.

After motivating the need for a notion of relevance in event query evaluation (Section 15.1), we give a precise definition of relevance and temporal relevance (Section 15.2). We then develop a method for statically (i.e., at compile time) determining temporal relevance, expressing it in the form of so-called temporal relevance conditions (Sections 15.3 and 15.4). During query evaluation (i.e., at run time), this enables garbage collection of events that become irrelevant as time progresses (Section 15.5). In addition, temporal relevance is also important at compile time for cost-based query planning. An outlook that discusses variations on the problem of determining temporal relevance (Section 15.6) and a discussion of related work (Section 15.7) complete this chapter.

15.1 Motivation: Garbage Collection, Query Planning

Consider the following XChange$^{\text{EQ}}$ program consisting of a single rule and a possible query plan QP for it.

```
DETECT
  c[var X]
ON
  and {
    event i: a[var X],
    event j: b[var X],
  }
  where {
    {i,j} within 2 hours
  }
END
```

$$QP = \langle \begin{array}{lll} A & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{a[var X]}](E) \\ B & := & \mathsf{Q}^{\mathsf{X}}[i : \texttt{b[var X]}](E) \\ C & := & \sigma[\max\{i.e, j.e\} - \min\{i.s, j.s\} \leq 2](\\ & & \quad A \bowtie B) \end{array} \rangle$$

When during the evaluation of this query an event `a[42]` is received at time t_1, then an appropriate tuple has to be stored in $\circ A$ since it might contribute to an answer to the query when an event `b[42]` is received at a later time $t_2 > t_1$.

While it is necessary to store received events for some time, it is for this query not necessary to store them forever. In our example, the condition that the a and b events happen within 2 hours of each other renders any a or b event tuple that is older than 2 hours irrelevant for future

answers to this specific query or rule. Since we have only this single rule, any stored event tuple in $\circ A$ or $\circ B$ can be removed after 2 hours.

We call this period of time for which an event or an intermediate result must (at least) be stored its *temporal relevance*. As we will see, determining temporal relevance becomes an involved problem when more complex queries than this example are considered and when we take into account an incremental evaluation that materializes intermediate results. The relevance of intermediate results is a noteworthy issue since it does not simply derive from the relevance of its constituent events.

Knowing how long an event (or an intermediate result) is relevant is a prerequisite for performing garbage collection during event query evaluation. Garbage collection removes events that have become irrelevant from their stores and thus frees up memory. It is important for event query evaluation for two reasons. First, if no garbage collection is performed, then the number of stored events always grows over time, usually at least linearly w.r.t. the size of the event stream received so far. When we assume an infinite stream of events, then we are thus guaranteed to run out of memory sooner or later. Second, garbage collection can be relevant even when the event stream is not infinite and running out of memory not a concern (e.g., in cases where event queries are part of a business or database transaction of limited duration). Because the search time for a stored event (e.g., as part of the evaluation of a join) grows with the number of stored events, garbage collection can contribute to the efficiency of event query evaluation by reducing the number of stored events.

Temporal relevance is also of central importance for developing cost-based query planners (akin to cost-based query planners found in traditional databases): an important input of any cost estimation function is the cardinality of event stores, which in turn is proportional to the length of time events are stored.

15.2 Temporal Relevance: Problem Definition

If we were to evaluate query plans incrementally as described in the previous chapter without any notion of temporal relevance, we would have to store the full history of all event tuples for each base relation and materialization point. Our goal with temporal relevance is to limit the stored history of event tuples to only the necessary knowledge of the past (as needed for producing all answers).

Temporal relevance is in this sense symmetrical to temporal preservation (cf. Chapter 13.3): temporal preservation makes a statement about what knowledge of the future is needed for query evaluation, temporal relevance about knowledge of the past. However they are asymmetrical in that no knowledge whatsoever is needed about the future, while a significant amount of knowledge about the past can be needed. Further, temporal preservation is a property that is always given (by design of CERA), while temporal relevance is specific to a given query plan and has to be determined in some algorithmic way.

15.2.1 Relevance and Temporal Relevance

The need to store histories of event tuples comes from joins in CERA expressions. Only joins combine event tuples received at different times. The relational algebra expression obtained through finite differencing (cf. Figure 14.2) $\triangle(R \bowtie S) = \circ R \bowtie \triangle S \ \cup \ \triangle R \bowtie \triangle S \ \cup \ \triangle R \bowtie \circ S$ shows this since it contains event histories ($\circ R$, $\circ S$). We can remove event tuples from the histories, when we know for sure that they will not affect any future answers, i.e., when they become irrelevant.

Definition Let E be a CERA expression with input relations $R_1, \dots R_n$. A tuple r of an input relation R_i is **relevant** for E at time *now*, if it might be joined (now or at a later time) with tuples from the other relations to produce an answer e to E with an occurrence time $m_E(e)$ of or later than *now*.

Query Plan 1:

$$\begin{array}{lll}
C & := & \mu[c \leftarrow a \sqcup b](\\
 & & \sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2](\\
 & & \quad A \bowtie B\ )) \\
F & := & \mu[f \leftarrow c \sqcup d \sqcup e](\\
 & & \sigma[c.e < d.s](\\
 & & \sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4](\\
 & & \sigma[d.e < e.s](\\
 & & \sigma[\max\{d.e, e.e\} - \min\{d.s, e.s\} \leq 1](\\
 & & \quad (C \bowtie D) \bowtie E\ )))))
\end{array}$$

Query Plan 2:

$$\begin{array}{lll}
C & := & \mu[c \leftarrow a \sqcup b](\\
 & & \sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2](\\
 & & \quad A \bowtie B\ )) \\
V & := & \sigma[c.e < d.s](\\
 & & \sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4](\\
 & & \quad C \bowtie D\ )) \\
F & := & \mu[f \leftarrow c \sqcup d \sqcup e](\\
 & & \sigma[d.e < e.s](\\
 & & \sigma[\max\{d.e, e.e\} - \min\{d.s, e.s\} \leq 1](\\
 & & \quad V \bowtie E\ )))
\end{array}$$

Figure 15.1: Example query plans with different materialization points

Note that the occurrence time of the answer implies that at least one of these other tuples has a time stamp $\geq now$. When a tuple joins with other tuples, it might actually not produce an answer for E. This is the case when the resulting tuple(s) of the join are eliminated by a selection in E.

Temporal relevance Temporal relevance is a particular form of relevance that is derived only from selections expressing temporal conditions and where we make no assumptions about what other tuples are currently stored or might arrive in the future. We discuss other, more involved forms of relevance in the outlook of Section 15.6

Example For our examples in this chapter, we will mainly use Rel^{EQ} instead of $\text{XChange}^{\text{EQ}}$. This preserves the essentials of temporal relevance but gives us the benefit of more compact notation and examples (e.g., because there are no pattern matching or construction operations). The concepts and solutions in this chapter are immediately applicable also to $\text{XChange}^{\text{EQ}}$, however. Consider the following example Rel^{EQ} program.

$$\begin{array}{lll}
\text{F}(x) & \leftarrow & c : \text{C}(x),\ d : \text{D}(x),\ e : \text{E}(x),\ c \ \textsf{before}\ d,\ \{c, d\}\ \textsf{within}\ 4,\ d\ \textsf{before}\ e,\ \{d, e\}\ \textsf{within}\ 1 \\
\text{C}(x) & \leftarrow & a : \text{A}(x),\ b : \text{B}(x),\ \{a, b\}\ \textsf{within}\ 2
\end{array}$$

Two possible query plans for this program are shown in Figure 15.1. The base relations of both query plans are A, B, D, and E, which contain the event tuples for $\text{A}(x)$, $\text{B}(x)$, $\text{D}(x)$, and $\text{E}(x)$, respectively.[1] Query plan 2 uses a materialization point V for evaluation of F, while query plan 1 does not.

For an example of events becoming irrelevant over time, consider the expression for C in Figure 15.1, which is the same for both query plans. Let $a = \{a.s = 9, a.e = 10, x = 42\}$ and

[1] Note that since we consider Rel^{EQ} here, we do not have a single stream of incoming events E here, but separate streams A, B, D, E for each type of event.

$b = \{b.s = 11, b.e = 11, x = 42\}$ be tuples in the event histores $\circ A$ and $\circ B$ of A and B, respectively. At time $now = 12$, tuple a is not relevant anymore: any tuple r resulting from joining a with a B-tuple arriving at this or a later time (i.e., having $b.e \geq now = 12$) will be eliminated by the expression's selection $\sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2]$ and thus not produce an answer. Tuple b is still (temporal) relevant at time $now = 12$, and also —to recall that the time domain is not limited to integers— at time $now = 12.3$. It becomes irrelevant when $now > 13$ due to the selection.

15.2.2 Temporal Relevance Conditions

Temporal relevance must be expressed in a formal way and in a restricted syntax so that we have (1) a clear notion of output of an algorithm that statically determines temporal relevance, (2) a basis for implementing garbage collection in query evaluation, and (3) means for correctness proofs. One might expect that temporal relevance could be expressed just as time window for each relation that states how long each tuple must be kept. However, since tuples have several time stamps —which is necessary for expressive event queries— this is not sufficient, and we express temporal relevance as so-called temporal relevance conditions. Conditions have also the advantage that they generalize well to other forms of relevance (cf. Sections 15.6.

For each materialization point Q in a given query plan QP, and each input relation R (base relation or materialization point) of Q, we will have a temporal relevance condition $TR_{R \text{ in } Q}$. If $TR_{R \text{ in } Q}$ is true for a tuple $r \in R$ at the current time now, then this tuple is still relevant. If it is false, the tuple is irrelevant and can be removed by the garbage collector. Removing is "optional," i.e., irrelevant tuples that are not removed do not disturb query evaluation. Temporal relevance conditions derived in this work have the following form:

$$\begin{aligned} TR_{R \text{ in } Q} \equiv \quad & i_1 \geq now - rt_1 \wedge \ldots \wedge i_m \geq now - rt_m \wedge \\ & i_{m+1} > now - rt_{m+1} \wedge \ldots \wedge i_n > now - rt_n \end{aligned}$$

where $\{i_1, \ldots i_n\} \subseteq sch_{time}(R)$ are some time stamps of R. All rt_k are fixed durations (lengths of time) and can be understood as individual time windows for each time stamp in the condition. A rt_k is called the relevance time of time stamp i_k, and also written $rt(i_k)$ or, making explicit the input relation R and the expression or materialization point Q, $rt_{R \text{ in } Q}(i_k)$. Note that the syntax of temporal relevance conditions is such that they can be used as a conditions of selection operators when we read now as a constant expressing the current time. This is convenient because garbage collection can be implemented using existing functionalities from selection operators.

15.2.3 Correctness Criterion

Definition Let QP be a query plan and $K = \{TR_{R \text{ in } Q}, \ldots\}$ a set of temporal relevance conditions, one for each input relation R of each materialization point $Q := E$. K is correct if for all possible base relations of QP the following holds for each time point now and each materialization point $Q := E$ of QP: all current (at time now) and future results of E are the same as those of E', i.e., $\sigma[M_E \geq now](E) = \sigma[M_{E'} \geq now](E')$, where E' is obtained from E by replacing each input relation R (base relation or materialization point) with any R' such that $\{r \in R \mid TR_{R \text{ in } Q}(r)\} \subseteq R' \subseteq R$.

Note that when the input relation R is a materialization point, then R' derives from the definition of $R := E_R$ in the *original* query plan QP, and *not* from some "E'_R" where in turn a replacing of the input relations of E_R has taken place.

The intuition of correctness is simple: the temporal relevance conditions are correct when removing tuples that are, according to the conditions, deemed irrelevant from the event histories (which corresponds to replacing the full history R with R') does not influence the query result at the current time now or in the future. We do not care about past results because they have already been produced in earlier evaluation steps. Note that since $\sigma[M_E \geq now](E) = \sigma[M_{E'} \geq now](E')$ must hold for all possible times now, we could equally write $\sigma[M_E = now](E) = \sigma[M_{E'} = now](E')$ in the definition.

Query Plan 1:

$$\begin{aligned}
TR_{A \text{ in } C} &\equiv a.s \geq now - 2 \wedge a.e \geq now - 2 \\
TR_{B \text{ in } C} &\equiv b.s \geq now - 2 \wedge b.e \geq now - 2 \\
TR_{C \text{ in } F} &\equiv c.s \geq now - 5 \wedge c.e \geq now - 5 \\
TR_{D \text{ in } F} &\equiv d.s \geq now - 1 \wedge d.e \geq now - 1 \\
TR_{E \text{ in } F} &\equiv e.s > now - 1 \wedge e.e \geq now - 0
\end{aligned}$$

Query Plan 2:

$$\begin{aligned}
TR_{A \text{ in } C} &\equiv a.s \geq now - 2 \\
TR_{B \text{ in } C} &\equiv b.s \geq now - 2 \\
TR_{C \text{ in } V} &\equiv c.s \geq now - 4 \\
TR_{D \text{ in } V} &\equiv d.s > now - 4 \wedge d.e \geq now - 0 \\
TR_{V \text{ in } F} &\equiv d.s \geq now - 1 \\
TR_{E \text{ in } F} &\equiv e.s > now - 1 \wedge e.e \geq now - 0
\end{aligned}$$

Figure 15.2: Temporal Relevance Conditions

In the following when we talk about determining temporal relevance conditions, we of course always mean implicitly that these conditions must be correct according to the definition just given.

Example The temporal relevance conditions for the query plans from Figure 15.1 are shown in Figure 15.2. The first four conditions of query plan 1 are longer than they need to be: the comparison $a.e \geq now - 2$ is superfluous since it is implied by $a.s \geq now - 2$ and the temporal conditions of the query expression (analogous for $b.e$ and $c.e$). For query plan 2, all conditions are minimal in the sense that they contain no superfluous comparisons. For both query plans, each condition is optimal in the sense that the time windows cannot be made tighter for any time stamp.

15.3 Determining Temporal Relevance Conditions

Our task now is to determine the relevance time $rt_{R \text{ in } E}(i)$ of each time stamp i of an input relation in a given expression E. We focus in this section on the general idea and give a full algorithm in the next section. Note that we determine the relevance times statically at the compile time of a query. Accordingly when we speak of a "time stamp" in the following, we always mean the element of the schema (e.g., $i.e$), not a concrete value of a given tuple (e.g., $r(i.e) = 42$). Since our method runs at compile time we do not have any concrete values but reason abstractly about all possible values.

15.3.1 Temporal Distances

For determining relevance times, we use an auxiliary computation. For each pair i, j of time stamps in the expression E, we establish from the temporal selections in E an upper bound $td(i, j) \geq 0$ such that any tuples in the result of E will obey $j - i \leq td(i, j)$. The upper bound depends on the temporal selections in E and we want it to be as small as possible. For reasons that will become clear shortly, we call this least upper bound the temporal distance from i to j. The temporal distances from i to j and from j to i can be different. For example in the expression of F_f of query plan 1, $td(c.e, d.s) = 4$ but, since c must happen before d, $td(d.s, c.e) = 0$.

For a given time stamp i, its relevance time $rt(i)$ then simply is the longest of all temporal distances $td(i, j)$ from i to some other time stamp j of E, i.e.,

$$rt(i) = \max\{td(i, j) \mid j \in sch_{time}(E)\}.$$

This equation for the relevance time can be explained as follows. Let the current time be *now*. Consider a time stamp i for a tuple r_1 that is stored in an input relation R_1 of expression E. Note that since the tuple is already stored, the value for the time stamp i is $r_1(i) < now$. Now let another tuple r_2 of another relation R_2 arrive at the current time *now*, i.e., $m_E(r_2) = now$. Let the two tuples join (together with other tuples from other relations in case there are more than two input relations) to yield a tuple e. Let $j \in sch_{time}(R_2)$ be a time stamp such that its value $r_2(j) = now$; since $m_E(r_2) = now$ such a j must exist. Suppose now that $r_1(i) < now - rt(i)$, i.e., r_1 is deemed irrelevant according to its relevance time. Then $r_2(j) - r_1(i) > rt(i)$, and thus also $e(j) - e(i) > rt(i)$. Construction of $rt(i)$ ensures that there must be a selection in E that eliminates e. Note that this selection does *not* necessarily involve either i or j.

15.3.2 Temporal Distance Graph

It turns out that all temporal conditions in selections that are of interest for determining temporal relevance are equivalent to a conjunction of comparisons of the form $j-i < t$ or $j-i \le t$, where i, j are time stamps. For example, $\max\{a.e, b.e\} - \min\{a.s, b.s\} \le 2$ (translated from `{a,b} within 2 hours`) is equivalent to $b.e - a.s \le 2 \wedge a.e - b.s \le 2 \wedge a.e - a.s \le 2 \wedge b.e - b.s \le 2$. Similarly, $c.e < d.s$ (from `c before d`) gives us $d.s - c.e < 0$. To simplify explanation, we assume for now that we only have conditions of the form $j - i \le t$. The extension to $<$ later is simple and given in Section 15.3.3.

This observation is very helpful because it allows us to frame the problem of computing all temporal distances in an expression E as an "all pairs shortest paths" (APSP) problem in a directed, weighted graph. We refer to the graph we construct as the *temporal distance graph* (TDG) of an expression E. Its nodes are all the time stamps occurring in E. Each temporal condition $j - i \le t$ in a selection of E generates a directed edge from node i to j with weight t.[2]

Further, for each pair $i.s, i.e$ of time stamps, we generate an edge with weight 0 from $i.e$ to $i.s$. These edges reflect that the ending time of an event cannot be before its starting time ($i.s \le i.e \iff i.e - i.s \le 0$). We will later see that edges will also be generated for information known about input relations or for merging of time intervals.

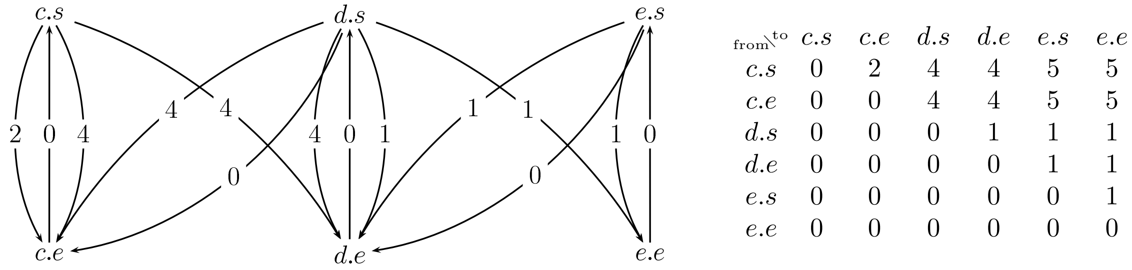
Since the temporal distance $td(i, j)$ between two time stamps is the least upper bound t so that $j - i \le t$, it corresponds to the (length of the) shortest path from i to j in the graph. The reason for this is that, like path length in a graph, bounds between time stamps obey the triangle equality, i.e., from $k - j \le t_1$ and $j - i \le t_2$ we can conclude that $k - i \le t_1 + t_2$.

The matrix of all shortest paths between all time stamps in E can be computed using a standard algorithm for solving the all pairs shortest paths (APSP) problem such as Floyd-Warshall [Sed90]. The relevance time $rt(i)$ for i then is simply the maximum entry in the row corresponding to i in this matrix.

Example Figure 15.3 shows the temporal distance graph for the expression of F of query plan 1 of Figure 15.1 and the corresponding matrix of shortest paths. Compare the maximum entries in each matrix row with the relevance times of Figure 15.2. The graph does not (yet) include nodes for the time stamps $f.s$ and $f.e$. It has an edge from $c.s$ to $c.e$ with weight 2, which reflects knowledge about input relation C. Details for this will be provided shortly. Note that it is not the general case that the lower-left half of the matrix is 0; this is only because this particular query demands the events c, d, e occur sequentially.

Having presented the general idea, the remainder of this section refines it with some necessary details, before the full algorithm is given in the next section.

[2] A mnemonic for the direction of the edge is as follows: The less-or-equal sign ($\le$) in $j - i \le t$ faces in the direction from i to j — that is the same direction that the arrow head of the directed edge in a drawing of the temporal distance graph would have. The intuition becomes even clearer if we reform $j - i \le t$ to the equivalent $i + t \ge j$, which we can read as "a tuple with time stamp i has to wait for tuples with time stamp j at most for t units of time."



from\to	$c.s$	$c.e$	$d.s$	$d.e$	$e.s$	$e.e$
$c.s$	0	2	4	4	5	5
$c.e$	0	0	4	4	5	5
$d.s$	0	0	0	1	1	1
$d.e$	0	0	0	0	1	1
$e.s$	0	0	0	0	0	1
$e.e$	0	0	0	0	0	0

Figure 15.3: Temporal distance graph and matrix shortest paths

$$
\begin{array}{ccccc}
(<,x) & + & (<,y) & = & (<,x+y) \\
(<,x) & + & (\leq,y) & = & (<,x+y) \\
(\leq,x) & + & (<,y) & = & (<,x+y) \\
(\leq,x) & + & (\leq,y) & = & (\leq,x+y)
\end{array}
\qquad
\begin{array}{ccccc}
(<,x) & < & (<,y) & \iff & x<y \\
(<,x) & < & (\leq,y) & \iff & x\leq y \\
(\leq,x) & < & (<,y) & \iff & x<y \\
(\leq,x) & < & (\leq,y) & \iff & x<y
\end{array}
$$

Figure 15.4: Strict and (non-)strict edges

15.3.3 Strict Inequalities

So far we have pretended that we have only non-strict inequalities such as $j - i \leq t$ and ignored the fact that some are in fact strict, i.e., of the form $j - i < t$. We suggest two ways to deal with strict inequalities.

First, we can simply treat all strict inequalities as non-strict when building the temporal distance graph, as we have done so far. (Of course, in the actual evaluation of the query plan, we will distinguish $<$ and $\leq$.) This is leads to a marginal "over-estimation" on the temporal relevance conditions and we will store in some cases more tuples than necessary. Since we only store *more* tuples, the result of the real evaluation is still correct. Because the difference between $\leq$ and $<$ amounts only for very few more tuples being stored, the consequences on performance are usually minimal.

Second, we can explicitly distinguish $<$ and $\leq$ by "extending" our time domain for durations $\mathbb{D}$ to $\{\leq,<\} \times \mathbb{D}$. The duration $(\leq, t)$ simply corresponds to the "old" t. Its sister duration $(<, t)$ can be imagined as $t - \epsilon$ with an infinitesimally small ϵ. Conditions such as $i - j < t$ can then be read as $i - j \leq (<, t)$. The operation $+$ and the relation $<$ on durations, which are needed in the computation of shortest paths and relevance times, must be adapted appropriately, as shown in Figure 15.4. Note that the relation $<$ automatically gives a definition for min and max operations on the extended time domain.

When we have determined the relevance time $rt(i)$ for a time stamp i (as maximum of i's row in the matrix), the first component ($<$ or $\leq$) in the tuple of will indicate whether the corresponding temporal relevance condition contains $i > \mathit{now} - w$ (in case $rt(i) = (<, w)$) or $i \geq \mathit{now} - w$ (in case $rt(i) = (\leq, w)$).

15.3.4 Recognizing Superfluous Relevance Times

In Figure 15.2, the first four temporal relevance conditions of query plan 1 contain superfluous comparisons: each right comparison is implied by the left. It is desirable for performance to avoid such unnecessary comparisons and have minimal temporal relevance conditions as those of query plan 2.

We can remove implied comparisons in a post processing. To do this, we must distinguish whether an edge in a temporal distance graph is already guaranteed to hold for all time stamps by the input relation (i.e., independently of the expression) or whether it has been added by some selection of the expression (and must be tested in the query plan evaluation). We will mark guaranteed edges with an exclamation mark (!) and non-guaranteed edges with a question mark (?). Now we can define when the relevance time of a time stamp covers the relevance time of

$$
\begin{array}{ccccc}
(?,x) & + & (?,y) & = & (?,x+y) \\
(?,x) & + & (!,y) & = & (?,x+y) \\
(!,x) & + & (?,y) & = & (?,x+y) \\
(!,x) & + & (!,y) & = & (!,x+y)
\end{array}
\qquad
\begin{array}{ccccc}
(?,x) & < & (?,y) & \iff & x < y \\
(?,x) & < & (!,y) & \iff & x < y \\
(!,x) & < & (?,y) & \iff & x \leq y \\
(!,x) & < & (!,y) & \iff & x < y
\end{array}
$$

Figure 15.5: Guaranteed and non-guaranteed edges

another time stamp to express that the comparison of the covered one is unnecessary.

Definition: The relevance time $rt(i)$ of a time stamp i is said to *cover* the relevance time $rt(j)$ of another time stamp j, if the shortest path from j to i in the temporal distance graph uses only edges marked as guaranteed and for its length $td(j,i)$ it holds that $td(j,i) + rt(i) = rt(j)$.

This can be explained as follows: Since the shortest path from j to i uses only guaranteed edges, $i - j \leq td(j,i)$ is true for any tuple in the input relation. Then, $i - j + rt(i) \leq rt(i) + td(j,i) = rt(j)$ (assumption) and $i \geq now - rt(i)$ together imply that $j \geq i + rt(i) - rt(j) \geq now - rt(j)$.

Commonly when solving the all pairs shortest path (APSP) problem we compute only the *length* of the shortest paths and not the paths themselves. The definition above however refers to the shortest paths itself to test whether they use only guaranteed edges. It turns out that we do not need the actual shortest paths but can make the information about guaranteed and non-guaranteed edges part of the time domain, much like we did with the $<$ and $\leq$ markings earlier. The corresponding definitions for addition and comparison are given in Figure 15.5. Note that the time domain that is extended can be already $\{\leq, <\} \times \mathbb{D}$ from earlier (Section 15.3.3).

15.3.5 Merging of Time Intervals

In the example of Figure 15.3, we have neglected time stamps $f.s$ and $f.e$, which are generated by the merging operation (μ) in F of query plan 1. One might be tempted to disregard such time stamps in the temporal distance graph since they do not occur in any of the temporal relevance conditions. However, (1) a temporal selection such as $\sigma[f.e - f.s < 1]$ could be performed after the merging and its effect should be propagated to the time intervals f has merged (in the example: c, d, e); and (2) knowledge about a time stamp generated by merging (e.g., c in the expression for C) might be needed in other expressions (e.g., $c.e - c.s < 2$ in F).

To include time stamps $j.s$ and $j.e$ generated by a merging $\mu[j \leftarrow i_1, \ldots i_n](E)$ into the temporal distance graph, we add them as nodes and add edges as follows: Since $j.s = \min\{i_1.s, \ldots, i_n.s\}$, it holds that $j.s - i_k.s \leq 0$ for $k = 1, \ldots, n$ and we add corresponding edges into the graph, which are marked as guaranteed (!). Analogously for $j.e = \max\{i_1.e, \ldots, i_n.e\}$, we add guaranteed edges for $i_k.e - j.e \leq 0$ ($k = 1, \ldots, n$). Further we add an edge from $j.s$ to $j.e$, its length being the longest of all temporal distances between any $i_k.s$ and $i_l.e$.

The edges between $j.s$ and the $i_k.s$, and between $j.e$ and the $i_k.e$ will propagate any temporal selection involving $j.s$ or $j.e$ "down" to the i_k. The length of the edge from $j.s$ to $j.e$ is justified by the definition of $j.s$ and $j.e$ as minimum and maximum. As will become clear in the full algorithm (cf. Section 15.4), the length can be computed immediately when $j.s$ and $j.e$ are added to the temporal distance graph since the merging operator discards the $i_k.s$ and $i_k.e$, and accordingly the edges between them are all already known.

15.3.6 Propagation in Query Plans

The temporal distance graph for a given expression E should make use of knowledge about input relations. An example is the constraint that $c.e - c.s < 2$ in the graph of Figure 15.3, which derives from the expression that computes the input relation C as a materialization point. When E has an input relation that is defined by a materialization point Q, the temporal distance graph for E should contain the subgraph of Q's temporal distance graph that contains only the nodes of $sch_{time}(Q)$ and edges between them. These edges should all marked as guaranteed (!) now in E's temporal distance graph since they are automatically satisfied.

Since query plans are acyclic, we can compute temporal relevance conditions from left to right so that Q's temporal distance graph is readily available when we have to compute the one for E.

15.3.7 Unions in Materialization Points

Recall that CERA does not have a unions, but that they can be expressed in materialization points of the form $Q := R_1 \cup \ldots \cup R_n$, where each R_i is either a base relation or a materialization point. We do not need to derive temporal relevance conditions for the input relations R_i in unions since there is no need to store them. However expressions referring to Q as an input relation need a temporal distance graph for Q (cf. Section 15.3.6.

The temporal distance graph for Q derives from the temporal distance graphs of the input relations $R_1, \ldots R_n$. The nodes are the time stamps of $sch_{time}(Q)$; note that $sch_{time}(Q) = sch_{time}(R_1) = \ldots = sch_{time}(R_n)$. Each edge from i to j in the temporal distance graph of Q is assigned the maximum of the individual lengths of the shortest paths between from i to j in the temporal distance graphs of the input relations $R_1, \ldots R_n$. Keep in mind that the length of this shortest path can be ∞, and an edge with weight ∞ is the same as having no edge.

Having unions only in this limited form at materialization points, not in CERA itself makes our explanations and the algorithm in the next section somewhat simpler. Consider an expression such as $Q := \sigma[\max\{i.e, j.e\} - \min\{i.s, j.s\} < 2](R_1 \cup R_2)$, which is not allowed in CERA. The temporal distance graph for it would contain only four time stamps $i.s$, $i.e$, $j.s$, $j.e$. However we should distinguish the time stamps of R_1 and R_2, since they can give different temporal relevance conditions. For example, R_1 might guarantee (from its definition as a materialization point) that $i.e < j.s$, while R_2 does not guarantee this. Then $TR_{R_1 \text{ in } Q} \equiv i.s \geq now - 2$ is correct, but for R_2 the relevance time of $j.s$ is not covered by that of $i.s$, i.e., $TR_{R_2 \text{ in } Q} \equiv i.s \geq now - 2 \wedge j.s \geq now$.

An alternative to restricting CERA as done here would be to distinguish time stamps of input relations, i.e., have eight time stamps $i_{R_1}.s$, $i_{R_2}.s$, ... in the example above. Making the adaption to this alternative is not hard. However, for the purpose of this thesis we found that it would make explanations and notation unnecessarily longwinded.

15.4 Algorithm

We now give an algorithm to compute all temporal relevance conditions for a given query plan QP. The algorithm is specified on a high abstraction level with mathematical functions. An implementation in a functional programming language could mirror the specification very closely. An implementation in an imperative language is also straight-forward (see also Chapter 16. Reader comfortable with the explanations of the previous section and less inclined to the algorithmic details may want to skip ahead to Section 15.4.4.

15.4.1 Computing the Temporal Distance Graph

Definition The temporal distance graph $tdg_{QP}(E)$ of an expression E in the context of a query plan QP is a directed graph $G = (V_G, E_G)$ with weighted edges. Its vertices are all time stamps occurring in E. Its edges have temporal distances (marked as strict or non-strict, and guaranteed or non-guaranteed) as weights, i.e., $E_G \subseteq V_G \times \mathbb{W} \times V_G$ with $\mathbb{W} = \{?, !\} \times \{<, \leq\} \times \mathbb{D}$. Note that we allow multiple edges with different weights between the same two nodes; however since we are eventually only interested in shortest paths, only the edge with the least weight is relevant.

Simplifying Assumption We assume for ease of presentation of the algorithm that E does not contain any renaming operations (ρ) on time stamps. For the query plans generated by the translation of the previous chapter, this is the case with the exception of the top-most renaming (which renames the result time stamps $r.s$, $r.e$ to $e.s$, $e.e$ so that they can become the input of other expressions). This top-most renaming can be easily ignored in computing the temporal

distance graph when taking it into consideration in the propagation of temporal distances between different expressions in the query plan.

In a similar manner, we assume that no relation or materialization point occurs more than once in the expression E. For query plans that have been translated using the improvement where every XChange^{EQ} pattern matching operation Q^{X} has its own materialization point (cf. Chapter 14.4.4) this is necessarily the case.[3]

An extension of our algorithm to properly deal with renaming of time stamps and multiple occurrences of the same relation in an expression is not hard. It requires to distinguish time stamps from different input relations and the output relation as has already been been discussed in Section 15.3.7. To keep explanations simple and short, however, we refrain from making this extension in the description here.

Auxiliary Functions We use two auxiliary functions. The length of the shortest path between nodes i and j in a temporal distance graph G is given with $sp(G, i, j) \in \mathbb{W}$. The addition operation and order relation on path lengths, which are necessary to define and compute shortest path lengths, have been given in Figures 15.4 and 15.5. The function g operates on edge weights, turning a non-guaranteed edge (?) into a guaranteed edge (!), i.e., $g(?, <, t) = (!, <, t)$, $g(!, <, t) = (!, <, t)$, etc.

Abstract Interpretation The computation of the temporal distance graph $tdg_{QP}(E)$ can be understood as an abstract interpretation (or pseudo-evaluation) [Cou96] of the expression E: the expression is evaluated in the same manner as usual, but on a different domain and with different interpretations of the operators. Instead of the standard domain (sets of tuples) we use a so-called abstract domain (temporal distance graphs) and instead of the standard interpretation of each operator (e.g., a join on relations) we use an abstract interpretation (e.g., for a join the union of graphs). Importantly, elements of the abstract domain correspond to ("are an abstraction of") sets of elements from the standard domain: a temporal distance graph corresponds to all relations whose tuples obey the restrictions set forth in the graph (e.g., if $(i, !, <, t, j)$ is an edge of $tdg(E)$ then in all possible results of E, all tuples obey $j - i < t$).

Keep in mind however that most of the literature on abstract interpretation focuses on the analysis of imperative programs, whereas we analyze relational queries. In comparison to abstract interpretation of imperative programs, computing temporal distance graphs is much simpler.

Like the standard interpretation of expressions, the temporal distance graph is defined inductively on the structure of expressions (with an additional case for unions of materialization points).

Base relations For a base relation R with $sch_{time}(R) = \{i.s, i.e\}$, there is a question what temporal distance dur_R should be assigned for the edge from $i.s$ to $i.e$. Without any further knowledge about a maximal duration for the base events in R we can simple let $dur_R = \infty$. However, it is often the case that base relations deliver "duration-less" events, i.e., events that happen at time points not over time intervals. (This is especially the case when the events arrive as messages and are assigned occurrence times by the evaluation engine not the event source.) In this case $dur_R = 0$. Any intermediate cases are also conceivable and covered by the algorithm.

We also create an edge from $i.e$ to $i.s$ with weight $(!, \leq, 0)$, which reflects that we always assume that $i.s \leq i.e$.

- $tdg_{QP}(R) = (V_G, E_G)$ for a base relation R with $sch_{time}(R) = \{i.s, i.e\}$, where
  $V_G = \{i.s, i.e\}$
  $E_G = \{(i.s, !, \leq, dur_R, i.e), (i.e, !, \leq, 0, i.s)\}$

[3] Note that even when the same simple event query (e.g., `a[var X]`) is used twice in a rule body, the two occurrences have different event identifiers (e.g., i and j) and accordingly lead to two different matching operations (e.g., $\mathsf{Q}^{\mathsf{X}}[i : \texttt{a[var X]}]$ and $\mathsf{Q}^{\mathsf{X}}[j : \texttt{a[var X]}]$).

The auxiliary base relations for timer events X_j are a special case of base relations. Since they define new time stamps $i.s$, $i.e$ relatively to time stamps $j.s$, $j.e$ of another event, they implicitly introduce conditions on the temporal distance between $i.s$, $i.e$, $j.s$, and $j.e$. We use a set E_X to express the edges generated from these constraints (an example follows).

- $tdg_{QP}(X_j) = (V_G, E_G)$ for an auxiliary timer event relation X_j with $sch_{time}(X_j) = \{i.s, i.e, j.s, j.e\}$, where
  $V_G = \{i.s, i.e, j.s, j.e\}$
  $E_G = \{(i.e, !, \leq, 0, i.s), (j.e, !, \leq, 0, j.s)\} \cup E_X$
  E_X is a set of edges reflecting the definition of X_j

The set of edges E_X reflecting the definition of X_j depends straightforwardly on the way that $i.s$ and $i.e$, the new time stamps that are defined in X_j, depend on $j.s$ and $j.e$, the time stamps of the anchor event that is used to define the relative timer event. For example for an auxiliary relation defined as

$$\begin{aligned} X_j =: \{x \mid \quad & (x(j.s), x(j.e)) \in \rho[j.s \leftarrow e.s, j.e \leftarrow e.e](\pi[e.s, e.e](E)), \\ & x(i.s) = x(j.s),\ x(i.e) = x(j.e) + 6\} \end{aligned}$$

we would have

$$E_X = \{(i.s, !, \leq, 0, j.s), (j.s, !, \leq, 0, i.s), (i.e, !, \leq, 0, j.s), (j.e, !, \leq, 6, i.e)\}.$$

Materialization Points The following three cases realize the propagation of results through materialization points and take care of unions in materialization points.

- $tdg_{QP}(Q) = (V_G, E_G)$ for a materialization point Q defined as $Q := E$ in QP (E not containing any renaming operation ρ), where
  $(V'_G, E'_G) = tdg_{QP}(E)$
  $V_G = sch_{time}(E)$
  $E_G = \{(i, g(w), j) \mid i \in V_G, j \in V_G, (i, w, j) \in E'_G\}$
- $tdg_{QP}(Q) = (V_G, E_G)$ for a materialization point Q defined as $Q := \rho[e.s \leftarrow r.s, e.e \leftarrow r.e](E)$ in QP with $sch_{time}(E) = \{r.s, r.e\}$, where
  $(V'_G, E'_G) = tdg_{QP}(E)$
  $V_G = \{e.s, e.e\}$
  $E_G = \{(e.x, g(w), e.y) \mid e.x \in V_G, e.y \in V_G, (r.x, w, r.y) \in E'_G\}$
- $tdg_{QP}(R_1 \cup \ldots \cup R_n) = (V_G, E_G)$, where
  $V_G = sch_{time}(R_1) = \ldots = sch_{time}(R_n)$
  $E_G = \{(i, m, j) \mid i, j \in V_G, m = \max_{k=1,\ldots,n} sp(tdg_{QP}(R_k), i, j)\}$

CERA Operators Finally, the following cases inductively define the temporal distance graph for a CERA expression. The ideas behind the individual cases have already been discussed in Section 15.3.

- $tdg_{QP}(E_1 \bowtie E_2) = (V_G^1 \cup V_G^2, E_G^1 \cup E_G^2)$, where
  $(V_G^1, E_G^1) = tdg_{QP}(E_1)$
  $(V_G^2, E_G^2) = tdg_{QP}(E_2)$
- $tdg_{QP}(\mu[j \leftarrow i_1 \sqcup \ldots \sqcup i_n](E)) = (V_G, E_G)$, where
  $(V'_G, E'_G) = tdg_{QP}(E)$
  $w = \max\{sp(tdg_{QP}(E), i_k.s, i_l.e) \mid k = 1, \ldots, n, l = 1, \ldots, n\}$
  $V_G = V'_G \cup \{j.s, j.e\}$
  $$\begin{aligned} E_G = \ E'_G \ & \cup \ \{(j.s, !, w, j.e), (j.e, !, \leq, 0, j.s)\} \\ & \cup \ \{(i_k.s, !, \leq, 0, j.s) \mid k = 1, \ldots, n\} \\ & \cup \ \{(j.e, !, \leq, 0, i_k.e) \mid k = 1, \ldots, n\} \end{aligned}$$

- $tdg_{QP}(E_1 \overline{\ltimes}_{i \sqsupseteq j} E_2) = (V_G^1 \cup V_G^2, E_G)$, where
  $(V_G^1, E_G^1) = tdg_{QP}(E_1)$
  $(V_G^2, E_G^2) = tdg_{QP}(E_2)$
  $E_G = E_G^1 \cup E_G^2 \cup \{(j.s, ?, \leq, 0, i.s), (i.e, ?, \leq, 0, j.e)\}$

- $tdg_{QP}(E_1 \ltimes_{i \sqsupseteq j} E_2) = (V_G^1 \cup V_G^2, E_G)$, where
  $(V_G^1, E_G^1) = tdg_{QP}(E_1)$
  $(V_G^2, E_G^2) = tdg_{QP}(E_2)$
  $E_G = E_G^1 \cup E_G^2 \cup \{(j.s, ?, \leq, 0, i.s), (i.e, ?, \leq, 0, j.e)\}$

- $tdg_{QP}(\mathsf{Q}^{\mathsf{X}}[i : t](E)) = (V_G, E_G)$, where
  $(V'_G, E'_G) = tdg_{QP}(E)$
  $V_G = V'_G \cup \{i.s, i.e\}$
  $E_G = E'_G \cup \{(i.s, !, \leq, 0, e.s), (e.s, !, \leq, 0, i.s), (i.e, !, \leq, 0, e.e), (e.e, !, \leq, 0, i.e)\}$

- $tdg_{QP}(\pi[X](E)) = tdg_{QP}(E)$

- $tdg_{QP}(\gamma[G, a \leftarrow F(A)](E)) = tdg_{QP}(E)$

- $tdg_{QP}(\sigma[C](E)) = tdg_{QP}(E)$ for a non-temporal condition C (i.e., a condition not involving time stamps)

- $tdg_{QP}(\sigma[i - j \leq t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, \leq, t, i)\}$

- $tdg_{QP}(\sigma[i - j < t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, <, t, i)\}$

- $tdg_{QP}(\sigma[\max\{i_1, \ldots, i_m\} - \min\{j_1, \ldots, j_n\} \leq t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j_k, ?, \leq, t, i_l) \mid k = 1, \ldots, n, l = 1, \ldots, n\}$

- $tdg_{QP}(\sigma[i \leq j](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, \leq, 0, i)\}$

- $tdg_{QP}(\sigma[i < j](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, <, 0, i)\}$

Note that for selection, the last three cases are just variations of the two cases before them and are given for the sake of completeness.

15.4.2 Computing Relevance Times and Temporal Relevance Conditions

Function $rt_{QP}(i, R, E)$ computes the temporal relevance time $rt_{R \text{ in } E}(i)$ of a time stamp $i \in sch_{time}(R)$ of an input relation R of an expression E or materialization point $Q := E$ in a given query plan QP:

- $rt_{QP}(i, R, E) = \max\{sp(tdg_{QP}(E), i, j) \mid j \in J\}$, where
  $R_1, \ldots, R_n$ are the input relations of E
  $J = sch_{time}(R_1) \cup \cdots \cup sch_{time}(R_n)$

Function $trs(QP)$ spells out all temporal relevance conditions for a given query plan QP in a straight-forward manner based on the relevance times. These temporal relevance conditions are still "maximal," i.e., contain also superfluous comparisons. It uses auxiliary functions $tr_{QP}(Q)$, which gives the temporal relevance condition of a single materialization point $Q := E$, and $comp(rt, i)$, which spells out the comparison in a temporal relevance condition corresponding to a given relevance time rt of a time stamp i.

- $trs(QP) = \bigcup_{Q:=E \in QP} tr_{QP}(Q)$
- $tr_{QP}(Q := E) = \{ \ TR_{R \text{ in } Q} \equiv \bigwedge_{i \in sch_{time}(R)} comp(rt_{QP}(i, R, E), i) \mid R \text{ is an input relation of } E \ \}$
- $comp(?, \leq, t, i) = i \geq now - t$
  $comp(?, <, t, i) = i > now - t$
  $comp(!, \leq, t, i) = i \geq now - t$
  $comp(!, <, t, i) = i > now - t$

15.4.3 Minimal Temporal Relevance Conditions

To obtain minimal temporal relevance conditions, which do not contain superfluous, "covered," comparisons, we define the boolean-valued function $covers_{QP}(R, E, i, j)$ in analogy of the definition in Section 15.3.4.

- $covers_{QP}(R, E, i, j) \iff sp(tdg_{QP}(E), j, i) = (!, \cdot) \ \wedge$
  $sp(tdg_{QP}(E), j, i) + rt_{QP}(i, R, E) = rt_{QP}(j, R, E)$

Note that there is, in some cases, not a single unique minimal storage condition. This occurs for example when two time stamps of an input relation are guaranteed to happen at the same time, i.e., $i = j$, and thus each covers the other. Unless both i and j are covered by some third time stamp, we can remove either but not both from the temporal relevance condition.

In practice this doesn't cause any problems, because it does not matter which one we remove and there usually is a natural tie breaker like the processing order of time stamps (e.g., in the for each loop below). Function $trmin_{QP}(Q := E)$ is the analog of $tr_{QP}(Q := E)$ with the difference that it gives a *minimal* storage condition. The set $S \subseteq sch_{time}(R)$ of time stamps where no time stamp in S covers another in S is used in its definition and can, e.g., be obtained with the given procedure.

- $trmin_{QP}(Q := E) = \{TR_{R \text{ in } Q} \equiv \bigwedge_{i \in S} comp(rt_{QP}(i, R, E), i) \mid R \text{ is an input relation of } E\}$
- Computation of S:
  $S \leftarrow \emptyset$
  For each $i \in sch_{time}(R)$:
  If $\neg \exists j \in S \ covers_{QP}(R, E, j, i)$
  $S \leftarrow (S \setminus \{j \in S \mid covers_{QP}(R, E, i, j)\} \cup \{i\}$

15.4.4 Complexity

The proposed algorithm can be implemented as a linear pass over the query plan with one step for each materialization point in it. Each step involves solving one all pairs shortest path problem. Using the Floyd-Warshall algorithm this can be done in time cubic in the number of time stamps occurring in the materialization point. Note that the intermediate shortest paths computation needed at a merging operator is part of solving the full all pairs shortest paths problem. Hence, it can be implemented in such a way that the work done there is saved in the later shortest paths computations and does not affect complexity.

While there are also subcubic algorithms for computing shortest paths, their associated overhead entails that they usually will not pay off in a practical implementation since the number of time stamps is too small (say < 20).

Our algorithm's complexity is easily acceptable because it is run only once during query compilation and because query compilation contains far more expensive operations. It might be interesting however, e.g., for branch and bound optimization of query plans, to investigate dynamic algorithms that can efficiently compute the effect of changes to an existing query plan on temporal relevance conditions or temporal distance graphs.

15.4.5 Correctness

The core ideas for a correctness proof have been given already in Section 15.3. By induction on the structure of expressions, one shows that the temporal distance constraints laid out by the computed temporal distance graph are satisfied by all possible results of an expression. That is, if the temporal distance graph contains an edge for $i - j < t$ or $i - j \leq t$ then all possible tuples in a result satisfy this. The argument of the end of Section 15.3.1 explains the correctness of the temporal relevance conditions derived from the temporal distance graph.

15.5 Using Temporal Relevance for Garbage Collection

Temporal relevance conditions give rise to garbage collection in our incremental query evaluation. This garbage collection can be done as part of each evaluation step. However, since irrelevant event tuples that have not been removed yet been from their event histories do no affect the result of query evaluation, garbage collection can also be performed in other ways. For example it might only be performed in regular intervals (e.g., every n-th evaluation step), be performed "on demand" (i.e., when memory gets scarce), or asynchronously in a separate thread that runs in parallel with the regular query evaluation.

15.5.1 Query Evaluation Algorithm with Garbage Collection

For illustration we describe here the simplest case where we perform garbage collection as part of every evaluation step of the conceptual algorithm described in Chapter 14.3.5. At the end of each step, we not only add the corresponding new events to each event history $\circ Q_i$, but also remove those events from $\circ Q_i$ that are irrelevant to all queries that access Q_i.

Note that there may be several temporal relevance conditions $TR_{Q_i \text{ in } Q_{j_1}}, TR_{Q_i \text{ in } Q_{j_2}}, \ldots$ for a given materialization point or base relation Q_i. This is the case when Q_i is used in multiple materialization point definitions $Q_{j_1} := E_{j_1}$, $Q_{j_2} := E_{j_2}$, ... of the query plan QP.

In the conceptual algorithm given here, we assume that each materialization point Q_i is associated with single event history ($\circ Q_i$). In principle it is however also conceivable to use several different event histories for a given materialization point. These different event histories might for example provide different index structures.[4] This could be useful when the materialization point is used in different expressions that perform joins on different attributes. Then it makes sense to have different event histories, where each provides an index for one of the different join attributes.

When we assume a single event history $\circ Q_i$ per materialization point Q_i, then tuples must be kept in $\circ Q_i$ as long as one of the temporal relevance conditions $TR_{Q_i \text{ in } Q_{j_1}}, TR_{Q_i \text{ in } Q_{j_2}}, \ldots$ for Q_i return true. Conversely, a tuple can be removed from $\circ Q_i$ when all temporal relevance conditions $TR_{Q_i \text{ in } Q_{j_1}}, TR_{Q_i \text{ in } Q_{j_2}}, \ldots$ for Q_i return false.

For the algorithm that follows let

$$TR_{Q_i} := \bigvee \{F \mid TR_{Q_i \text{ in } Q_j} \equiv F \in trs(QP)\}$$

[4] Note that an event history in this context is only the interface though which the evaluation of a given expression (after finite differencing) accesses the contents of a materialization point. Therefore the event history might just be an index that does not also store the event tuples physically.

be the disjunction of the temporal relevance conditions $TR_{Q_i \text{ in } Q_{j_1}}, TR_{Q_i \text{ in } Q_{j_2}}, \ldots$ for Q_i in the query plan QP. (Recall from Section 15.4.2 that $trs(QP)$ is a function computing the temporal relevance conditions of all materialization point definitions $Q := E$ in a given query plan QP.) As just explained, if TR_{Q_i} is true for a tuple in $\circ Q_i$, then it must be kept; if it is false, the tuple can be removed from $\circ Q_i$.

The evaluation of QP with garbage collection then follows the following schema, which extends the conceptual algorithm from Chapter 14.3.5 by adding garbage collection.

```
∘B1 := ∅; ... ; ∘Bm := ∅;
∘Q1 := ∅; ... ; ∘Qn := ∅;

while(true) {
    advance now to the occurrence time of the next incoming event(s);
    △Q1 := ∅; ... ; △Qn := ∅;
    initialize △B1, ..., △Bm with the current events;
    compute △Q1, ..., △Qn according to △QP;
    for i := 1...n {
        ∘Qi := σ[TR_Qi](∘Qi ∪ △Qi);
    }
    output △Q1, ..., △Qn;
}
```

Note that $Q_i := \sigma[TR_{Q_i}](\circ Q_i \cup \triangle Q_i)$ could equivalently be written as

$$\circ Q_i := (\circ Q_i \setminus \sigma[\neg TR_{Q_i}](\circ Q_i)) \cup \triangle Q_i,$$

which reflects an efficient evaluation of that expression in the form of an update more closely.

Note that TR_{Q_i} could be simplified with some temporal reasoning. For example when $TR_{Q_i} \equiv i.s \geq now - 5 \wedge i.s \geq now - 7$, then we can just use $TR_{Q_i} \equiv i.s \geq now - 7$ instead because the first condition is implied by the second.

15.5.2 Remark on Index Structures for Garbage Collection

An important aspect for implementing garbage collection efficiently is to quickly search and remove irrelevant tuples, i.e., tuples satisfying $\neg TR_{Q_i}$, in $\circ Q_i$. Appropriate index structures can help here greatly. A simple, but efficient index structure might just keep tuples sorted by one of the time stamps used in the temporal relevance conditions. Note that often there is only one such time stamp after the temporal relevance conditions have been minimized. Even when there are multiple time stamps, garbage collection might simply designate one to be used, ignoring the others at the cost of keeping sometimes more tuples in event histories then strictly necessary.

15.6 Outlook: Variations and other Forms of Relevance

Relevance of stored events and intermediate results in complex event queries is an issue that has been given only little consideration before (cf. also Section 15.7). Therefore it is worth discussing it beyond the concrete solution for determining temporal relevance conditions in XChange$^{\text{EQ}}$ that has been given in this chapter. We now broaden the scope and discuss variations on and generalizations of relevance.

15.6.1 Multiple Time Axes

In the presentation of the algorithm for determining temporal relevance as well as in our operational semantics in general, we have made the assumption that base events have two time stamps for start and end. As discussed in Chapter 2.4.4, however, there are situations in distributed systems where events are time stamped according to multiple time axes (e.g., one time axis for the clock

of the event source and another for the clock of the event receiver). Our operational semantics can be easily extended to accommodate such multiple time axes. Because time stamps are largely treated like regular attributes, the main thing that would be required are additional time stamp attributes in the base events.

When such an extension is made to the operational semantics, the algorithm for determining temporal relevance of this chapter is still applicable. Only the case for base relations has to be changed to include the additional time stamps of the further time axes. Edges between the time stamps of different time axes in the same event could express relations between the different time axes. For example, in a distributed system that needs multiple time axes because clocks cannot be perfectly synchronized, there will typically be a bound on the drift between different clocks. When two time stamps $i.s$ and $j.s$ are conceptually the same time but given according two two different clocks, then this maximal drift between the clocks is the temporal distance between $i.s$ and $j.s$. Accordingly, we can add edges in both directions between $i.s$ and $j.s$ with the drift bound as weight in the temporal distance graph. (The same goes for $i.e$ and $j.e$.)

15.6.2 Generalization of Temporal Relevance

We have made the assumption that all temporal conditions can be written as a conjunction of comparisons of the form $i - j < t$ or $i - j \leq t$. This covers a significant, and for many practical applications sufficient, spectrum of possible temporal conditions. In particular it covers all temporal conditions described in the model theory of XChange$^{\text{EQ}}$(see Figure 9.3).[5] Not covered are, for example, conditions involving periodic time intervals such as "workday" (defined, e.g., as Monday through Friday, 9am to 5pm, except holidays) which are often also based on domain-specific calendars (see also Chapter 6.8).

Our algorithm can still be used by "approximating" these conditions. For example, `{i,j} within workday` implies that `{i,j} within 8 hours` if "workday" is defined as above. While always correct, this approximation can be crude: an event received at 4pm will be stored for 8 hours instead of just 1 hour.

An alternative is to keep the general framework laid out in this article, but use a more advanced form of temporal reasoning. For example, one could allow not only numeric edge weights in the temporal distance graph but also symbolic ones like "workday." The notion and computation of the temporal distances (shortest paths) and the longest temporal distance (longest shortest path) must be adapted then. An important challenge is that calendrical notions are not generally comparable and summable like the numeric edge weights were, e.g., "workday" and "1 hour" cannot be compared or added easily. Research on temporal reasoning [FGV05] will provide an extensive foundation for answering problems like this; note however that the particular problem that must be solved for temporal relevance (establishing temporal distances) is a not a standard problem.

15.6.3 Dynamically Changing Query Plans

We have assumed the query plan to be static, i.e., the set of queries does not change at runtime. Removing queries from a query plan at runtime is without problems — with removing the query, we also remove all its relations (if they are not needed by other queries) and the corresponding relevance conditions. Adding a new query can be problematic, however, depending on the semantics of adding a query.

In the simplest case, a new query that is added at a time point t_0 "sees" only events happening *after* t_0. Any result tuples that are generated using incoming events before t_0 are not considered part of the query's answer. This situation is fairly similar to the static case, and all we have to do for the new query is to determine its relevance conditions and add them to the existing relevance conditions.

[5] The condition for `{i,j} apart-by d` cannot be expressed as a conjunction of comparisons of the form. However, this condition has no impact on temporal relevance because it does not help us in limiting the time events have to be stored in any way.

In the most general case, a new query is supposed to see also all events that happened *before* its adding at time t_0. If there is no knowledge about the queries that can be added, no event can ever be deleted because it always might be needed by some query that is added in the future. This makes all events relevant, and there is no real need for relevance conditions.[6] It should be noted that this general case can be considered unsolvable because it implies maintaining a history of events that grows at least linearly with the length of the (potentially infinite!) event stream.

We can also consider an in-between case where a new query can see events that happened *before* it adding at time t_0, but the query is restricted some way. The restriction can be, e.g., that only some specific events during a finite time windows $[t_0 - w, t_0]$ will be relevant for answering it. This restriction could be included into the static determination of relevance conditions by including it into the original query plan, e.g., as a kind of "artificial query" that simulates the restrictions on queries that can be added in the future. The artificial query then expresses a kind of "worst case" for any future queries.

In another in-between case, we can use our method to determine whether adding a given new query (which is supposed to see events before its t_0) is possible. For this we would determine the relevance conditions of the new query Q and then compare it with the existing relevance conditions to see if all required event are available.

Note that the issues mentioned here are inherent to the semantics and evaluation of event queries and exist independently of any method for determining temporal relevance.

15.6.4 Joins in Temporal Relevance Conditions

Our temporal relevance conditions use only comparisons of the form $i > now - t$ or $j \geq now - t$. More expressive temporal relevance conditions, e.g., containing joins with other relations, could in some cases allow to remove more events. Consider $TR_{C \text{ in } F}$ of query plan 1 (Figures 15.1 and 15.2). A tuple $r \in C$ that is older than 4 hours should already have a corresponding tuple $r' \in D$ that will join with it. The selection $\sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4]$ will eliminate join results produced with D-tuples arriving after more than 4 hours of (the starting time of) this $r \in C$. The temporal relevance condition for C could therefore be stronger, e.g., using a strawman notation akin to nested queries:

$$\begin{aligned} TR'_{C \text{ in } F} \quad \equiv \quad & c.s \geq now - 5 \wedge c.e \geq now - 5 \wedge \\ & (c.s < now - 4 \Rightarrow x \in \pi[x](\sigma[d.s \geq now - 4](D)) \end{aligned}$$

The nested query of "$x \in \ldots$" in the condition leads to a (semi-)join with relation D in the evaluation.

Such a join in relevance conditions is rather undesirable. It is expensive to perform, in particular since the intermediate result is not materialized and used in the query evaluation. We can achieve a similar effect, however, by using a different query plan that materializes the join of C and D. An example is query plan 2 of Figure 15.1; its $TR_{V \text{ in } F}$ has the same effect as the join of the above $TR'_{C \text{ in } F}$. As an additional benefit, the join is materialized and thus used in the query evaluation. Testing the relevance condition therefore does not add any overhead.

Note that the choice which query plan to use, i.e., in particular which intermediate results to materialize, is the responsibility of the query planner. Relevance times are an important input for its cost metric (cf. Section 15.1), and the cost metric should take into account not only the cost of producing query results but also of garbage collection, i.e., of evaluating temporal relevance conditions and removing tuples from their stores.

15.6.5 General Relevance

The (temporal) relevance conditions derived in this paper rely only on temporal conditions and we have made no assumptions on the contents of event streams, in particular their unknown future

[6] Even in such a scenario, relevance conditions can still be interesting though, e.g., if only those events that are needed for the current queries should be kept in main memory while a history of all events is maintained in secondary memory.

content (cf. the definition of temporal relevance in Section 15.2.1). There are also other ways to determine the relevance of event tuples than temporal conditions; we call this relevance also "general relevance" to contrast it with (purely) temporal relevance.

On particularly interesting problem of determining relevance is using axiomatic knowledge about the future content of event streams; we also call this "axiomatic relevance." Such axiomatic knowledge can be stated explicitly as rules or derived implicitly from other sources such as business process descriptions. Axioms about event streams are similar to integrity constraints in databases in that they limit possible contents of the event streams. They are however differently used: In databases integrity constraints are *checked* to avoid or react to updates that violate them. In determining axiomatic relevance, the axioms are in contrast *assumed to be true* without checking them.

A vivid example of axiomatic knowledge about event streams are constraints on key references between event streams, which are natural in many applications: In a typical shipping application, a "shipping notice" event must be followed by either a single "delivery" event or a single "return to sender" event. A tracking identifier acts as a key that correlates these events. In event queries involving a join between "shipping notice" events and either "delivery" or "return to sender" events, "shipping notice" event tuples become irrelevant (and can be removed) as soon as they have found their unique join partner. Without the constraint, the event tuple would have to be stored in case further join partners arrive.

In [BSW04], it is discussed how such a constraint speeds up query evaluation in data streams. Note however that there, the constraints are dynamically "mined" from the data streams not a priori specified as axioms.

15.7 Related Work

The problem of determining temporal relevance introduced and solved in this chapter has been given little consideration in work on complex event queries before. In part, this can be attributed to characteristics of earlier languages for complex event queries, which are restricted in comparison to XChange$^{\text{EQ}}$ (see also Chapter 3).

Composition Operators Most composition-operator-based event query languages do not have temporal restrictions that would allow garbage collection through temporal relevance (e.g., [GJS92a, GD93, CKAK94, AC06]). In part, this is due to their origin in active database systems research, where event queries are often assumed to run inside (short-lived) database transactions. Outside of transactions, however, bounds on the "life-spans" of events and garbage collection become important [BZBW95].

Some composition-operator-based languages therefore allow to add a temporal window to an operator or a subexpression (e.g., [ZS01, HV02, CL04, BEP06a]), e.g., $(A;B)_{1h}$ to indicate that B must follow A within 1 hour (see also Chapter 3.2.4). The life-spans in the language of Amit [AE04] are also similar to this. Since the time window is applied to the full subexpression, determining temporal relevance of the involved events in trivial. However, this specification of a time window is very limiting, and many temporal conditions —like those of the query for F(x) in Section 15.2.1— cannot be expressed with it.[7] Further, rewriting such event algebra expressions for the purpose of query optimization is far more constrained [CL04] than for a relational algebra variant like CERA.

In addition, some composition-operator-based languages offer event instance consumption and selection, which also affects the relevance of events and allows for garbage collection (see also Chapter 3.2.7). These mechanism are however very different and have in contrast to temporal relevance a direct impact on query semantics.

Data Stream Systems Query languages for data streams such as CQL [ABW06] typically require users to explicitly specify a time window with each input event stream (as part of the

[7] Note that $((C;D)_4;E)_1$ does not express the conditions of the query under interval-based semantics since the outer conditions entails that C and E happen within 1 hour not as required D and E.

stream-to-relation operation; see also Chapter 3.3.2). For base relations, such time windows loosely correspond to temporal relevance conditions (although they usually are only for one time stamp).

The main difference is that in our work, temporal relevance conditions are derived automatically from queries, while in data stream query languages temporal windows must be explicitly specified and affect query semantics. This is difference remains also when other types of windows in data stream systems such as predicate windows [GAE06, KLG07] are considered.

In Chapter 3.3.4, we have seen an example that determining the temporal windows in data stream languages is not easy when joins are involved. It essentially requires a similar reasoning about temporal distances by the programmer as the algorithm given in the chapter does automatically. Note also that data stream query languages usually have only very limited support for temporal conditions beyond the specification of such time windows, especially compared to XChange$^{\text{EQ}}$.

Production Rule Systems As discussed in Chapter 3.4, production rules are also often used to detect complex events, though they are not dedicated event query language. Events (and states) are asserted as facts and production rules then derive and assert new facts, e.g., complex events, or retract facts. Garbage collection, that is, retracting facts of events that have become irrelevant (to a given set of production rules), must traditionally be programmed manually, e.g., by writing appropriate rules to retract them. Programming errors in this garbage collection are easy to make and will cause incorrectness of query results or memory leaks.

An automatic garbage collection, as enabled by temporal relevance investigated in this work, is clearly preferable. In [WBG08] the production rule system Drools is extended with an automatic garbage collection of facts for events that have become irrelevant. The approach for determining how long events are relevant is similar to the temporal distance graph that has been used in this chapter.[8]

The approach presented in this chapter is more general than [WBG08] in some aspects. The automatic garbage collection of [WBG08] is done in the context of RETE networks, which have a fixed strategy for materializing intermediate results, and it relies on a left-deep join tree structure in RETE networks. Our approach is more flexible in that we consider more general query plans that support different materialization points (as discussed in Chapter 14.6.1) and have no restrictions on structure of CERA expressions and materialization points.

Further, garbage collection in [WBG08] works only by removing event facts from alpha nodes, which loosely correspond to event histories of base relations in our approach. Intermediate results, which are stored in the beta nodes of the RETE network, are only removed when a corresponding fact is removed from an alpha node. Our approach is different here. Garbage collection in base relations and materialization points is done independently. This allows for a more aggressive garbage collection. It is often possible to remove an event tuple from the history of a base relation while some tuples in histories of materialization points that have been generated using this base event tuple must still be kept. An example of this can be seen in query plan 2 of Figure 15.1: A C event tuple has to be kept in $\circ C$ only for four hours (cf. also the temporal relevance conditions in Figure 15.2), while a V event tuple that has been generated from this C tuple (through joining it with a D tuple) has to be kept longer in $\circ V$, namely for five hours.

[8] Note [WBG08] has been published at the same time as [BE08a], which first presented the approach described in this chapter.

Chapter 16

Proof-of-Concept Implementation

The operational semantics that have been developed in the previous chapters are accompanied by a proof-of-concept implementation of XChange^{EQ}. This prototype implementation is written in Java and available open source.

This chapter describes the current implementation of the XChange^{EQ} prototype, which realizes the core concepts of the event query language XChange^{EQ}. We first explain how the XChange^{EQ} prototype can be used for querying events (Section 16.1). Then we focus on its actual implementation, explaining how to build the prototype from its source code (Section 16.2) and providing an overview of the source code (Section 16.3). Finally we discuss some of the current limitations of the prototype (Section 16.4).

16.1 Using the XChange^{EQ} Prototype

The XChange^{EQ} prototype interacts with the outside world through a Java object that is called an "XChange^{EQ} engine." An engine evaluates a given XChange^{EQ} program. Note that there can in principle be multiple engines that evaluate different XChange^{EQ} program or even multiple engines for the same XChange^{EQ} program. The XChange^{EQ} program is passed to the engine object during the object creation (i.e., in the constructor).

After the creation, events can be sent to an XChange^{EQ} engine and the engine will evaluate the queries of it XChange^{EQ} program and trigger reactive rules in the program.

Providing the XChange^{EQ} prototype as a Java object gives great flexibility on how it can be used. For example it can be used just as an ordinary object that runs in the same execution thread as the sender of the events, it could be run in a separate thread, or it could be wrapped into a Web service.

16.1.1 Manual Stratification in XChange^{EQ} Programs

The current XChange^{EQ} prototype has some limitations with regards to the syntax of XChange^{EQ} program. While we will discuss limitations in detail in Section 16.4, an important restriction should be mentioned here already: XChange^{EQ} programs must be hierarchical (i.e., free of any recursive cycles, cf. Chapter 10.1.3) and explicitly divided into strata, that is, the programs must indicate which rules belong together in a stratum and how the strata are ordered. This division into strata can be done manually be the programmer; it is however also no big challenge to write a preprocessor that accepts a program without explicit division into strata, confirms that it is hierarchical, and derives an explicit stratification (cf. Section 16.4.1).

To this end, the prototype requires that the rules that are to be grouped together in one stratum are in textual block that is surrounded by the keywords `STRATUM` and `END`. The textual order of these blocks in the program indicates the order of the strata, with the lowest stratum (i.e., the stratum that operates only on incoming events not on any derived events) first.

```
STRATUM
    DETECT
        c { all var X }
    ON
        and {
            event a: a{{ var X }},
            event b: b{{ var X }}
        }
        where { {a,b} within 10 sec }
    END

END

STRATUM
    DETECT
        f { all var X }
    ON
        and {
            event c: c{{ var X }},
            event d: d{{ var X }},
            while c: not e { var X }
        }
        where { {c,d} within 20 sec, c before d }
    END
END

STRATUM
    RAISE
        to(recipient="EngineUsageExample.raised", transport="java") {
            result { var X }
        }
    ON
        event f: var X -> f {{ }}
    END
END
```

Figure 16.1: Example an XChange^{EQ} program with "manual" stratification

Figure 16.1 shows an example of an XChange^{EQ} program that is annotated with the subdivision of its rules into strata as required by the current prototype. We will use this example program in the following, assuming that is in a file named `example.xchange`.

16.1.2 Simple Usage Example for an XChange^{EQ} Engine

There are several forms of XChange^{EQ} engines that differ in the control they provide over the time stamps that are assigned to events received by the engine. We start with the simplest case where events are time stamped by the engine upon reception with the current system time. The class for engine objects of this kind is called `TimeStampingXChangeEQEngine`.

Figure 16.2 shows a Java program that illustrates the use of the `TimeStampingXChangeEQEngine` with the example XChange^{EQ} program by generating some events. The main method first creates a new engine object using the example XChange^{EQ} program `example.xchange`. Then it sends the engine an event with the data term `a{"1", "2", "3"}`. Approximately one second later it sends the engine the event `b{"2", "3", "4"}`. (The call `Thread.sleep(1000)` causes the execution to pause for approximately 1000 milliseconds. Note that the two lines after this call are commented out.) Another three seconds later it sends the engine the event `d{"3"}`.

At this point, the reactive rule in the XChange^{EQ} program of Figure 16.1 will fire and call the method `raised` in our Java program. As explained in Chapter 6.5, the method receives as parameter the XML document constructed in the rule head of the reactive rule. The method will then simply print this XML document on the console.

When our Java program is executed, it will produce the following output:

```
import org.w3c.dom.Document;

import xchangeeq.TimeStampingXChangeEQEngine;
import xchangeeq.commons.Util;

public class EngineUsageExample {
    public static void main(String[] args) throws InterruptedException {
        TimeStampingXChangeEQEngine engine =
            new TimeStampingXChangeEQEngine("example.xchange");

        System.out.println("Starting...\n");
        System.out.flush();
        engine.stepWithTermAsString("a{ \"1\", \"2\", \"3\" }");
        Thread.sleep(1000);
        // engine.stepWithTermAsString("e{ \"3\" }");
        // Thread.sleep(1000);
        engine.stepWithTermAsString("b{ \"2\", \"3\", \"4\" }");
        Thread.sleep(3000);
        engine.stepWithTermAsString("d{ \"3\" }");
        System.out.println("End.");
        System.out.flush();
    }

    public static void raised(Document doc) {
        System.out.println(Util.xmlToString(doc));
        System.out.flush();
    }
}
```

Figure 16.2: Example for using the `TimeStampingXChangeEQEngine`

```
Starting...

<result>
<f>3</f>
</result>

End.
```

Note that there is a pause of roughly four seconds between the first line and the rest of the output.

We can also uncomment the two lines that are commented out in of Java program of figure 16.2. In this case, the engine will receive an event `e{"3"}` between event `a{"1", "2", "3"}` and `b{"2", "3", "4"}`. The effect of this is that the event query of the second rule in our example XChange^{EQ} program of Figure 16.1 will not have an answer due to the negation. Accordingly, the reactive rule will not be triggered, the `raise` method in our Java program not be called, and the output just be:

```
Starting...

End.
```

There will be a pause of roughly five seconds between the first line and the rest of the output.

16.1.3 Different XChange^{EQ} Engines and Event Time Stamps

The current XChange^{EQ} prototype supports three different engines that differ mainly in the control they give the user over the occurrence times of events.

Methods in `TimeStampingXChangeEQEngine` The `TimeStampingXChangeEQEngine` provides the following methods for receiving events (all with return type `void` and visibility `public`):

- `stepWithTermAsString(String termAsString)`

- `stepWithTerm(Term term)`
- `stepWithXML(Document doc)`
- `stepWithTermsAsStrings(List<String> termsAsStrings)`
- `stepWithTerms(List<Term> terms)`
- `stepWithXMLs(List<Document> docs)`

In the parameters of these methods, the type `Term` is from the package `xchangeeq.minixcerpt` (which belongs to the XChange^{EQ} prototype) and represents an Xcerpt data term. The type `Document` is from the package `org.w3c.dom` (which is usually provided by the Java distribution) and represents an XML document through the Document Object Model (DOM) [H$^+$08].

The first three methods are for sending a single event to the engine, the other three for sending several events to the engine. Whenever one or more events are sent to the engine, the engine time stamps these events with the current system time and preforms an evaluation step for the event queries in the engine's XChange^{EQ} program. Note that multiple calls to `stepWithTermAsString` with one event each are not equivalent to a single call to `stepWithTermsAsStrings` with a list of all the events. The reason for this is that in the former case the events may receive slightly different time stamps while in the latter case all events are assigned the same time stamp. (The same is true for the other corresponding pairs of methods.)

An events can be sent to the engine as a String that is a textual representation of an Xcerpt data term, as a data term object (type `Term`), or as an XML document (type `Document`).

More flexibility with `XChangeEQEngine` There are cases in event processing where the time stamp of an event should be determined by the event source not the recipient of the event as is the case with the `TimeStampingXChangeEQEngine`. To this end, the XChange^{EQ} prototype offers a more flexible engine called `XChangeEQEngine`. With this engine, the control for time stamps of events is outside the engine. `XChangeEQEngine` supports the following methods (again all with return type `void` and visibility `public`):

- `addEventAsTermAsString(TimePoint start, TimePoint end, String termAsString)`
- `addEventAsTerm(TimePoint start, TimePoint end, Term term)`
- `addEventAsXML(TimePoint sStart, TimePoint end, Document doc)`
- `step(TimePoint now)`

The first three methods are for sending events to the engine. In contrast to the methods of `TimeStampingXChangeEQEngine`, these methods of `XChangeEQEngine` allow to assign an occurrence time to the events. The occurrence time is a time interval represented as start and end time points, which are objects of the class `TimePoint` (of package `xchangeeq.time`). We discuss this class further down.

Also in contrast to the methods of `TimeStampingXChangeEQEngine`, the methods of `XChangeEQEngine` do not immediately cause query evaluation. Query evaluation must be initiated with a call to `step` giving the current time as parameter.

Note that these methods give the user a very fine grained control over the event query evaluation. Accordingly, some care is necessary with respect to the timing of events. When several events are sent to the evaluation engine between evaluation steps, they must all have the same end time. Only after a call to `step` that has this end time as parameter, events with a later end time may be added. Note that adding events out of order (i.e., adding an event with an earlier end time that the time of the last step) is not allowed.

Simplification with SimplifiedXChangeEQEngine It is common in event processing that the base events that will be received by the XChange^{EQ} engine occur only at time points rather than over (true) time intervals. The XChange^{EQ} prototype therefore offers also a simplified engine that has methods for only receiving such time point events. It is called SimplifiedXChangeEQEngine and has the following methods:

- stepWithTermAsString(TimePoint now, String termAsString)
- stepWithTerm(TimePoint now, Term term)
- stepWithXML(TimePoint now, Document doc)
- stepWithTermsAsStrings(TimePoint now, List<String> termsAsStrings)
- stepWithTerms(TimePoint now, List<Term> terms)
- stepWithXMLs(TimePoint now, List<Document> docs)

As with the TimeStampingXChangeEQEngine, receiving events and initiating query evaluation is done in just one method call (not separate methods calls as in the XChangeEQEngine). In contrast to TimeStampingXChangeEQEngine, the occurrence time is assigned by the sender of the event not the engine. Again, care is necessary so that these methods are called only with strictly increasing time stamps.

Creating TimePoint objects Time points are represented with objects of the abstract class TimePoint. The XChange^{EQ} prototype supports in principle different time domains. For time points that are based on Java's standard representation for date and time (Date class in package java.util), the concrete subclass RealTimePoint is used. The constructor of this class takes a Date object for the time point or alternatively the number of milliseconds elapsed since the epoch (midnight, 1 January 1970) as long integer.

The XChange^{EQ} prototype usually works with RealTimePoint to represent time points. However it also provides a class UnitTimePoint that models the time domain simply as integers. These UnitTimePoints is mainly useful for testing and debugging of the XChange^{EQ} prototype and not so much for practical applications.

16.2 Building XChange^{EQ} from the Source Code

We now describe how the XChange^{EQ} prototype can be built from its source code.

16.2.1 Obtaining Source Code

The source code of the current version of the XChange^{EQ} prototype is available in its Subversion repository [Sub], which is located at:

https://svn.cip.ifi.lmu.de/~eckert/svn/xchangeeq/

An archive of the source code is also available from

http://www.pms.ifi.lmu.de/projekte/xchangeeq/

16.2.2 Requirements

To build the prototype, Java and the Java SE Development Kit (JDK), Version 1.6.0 [Sun] are required. Additionally, the following freely available tools and libraries are needed:

- ANTLR Parser Generator, Version 3.0.1 [ANTb, Par07], which is used to generate the XChange^{EQ} parser from a grammar description.

- Java Universal Network/Graph Framework (JUNG) 1.7.6 [JUN], which is use to display algebra expressions as tree (useful mainly for debugging purposes).
- Commons-Collections [Com], which is required by JUNG.
- CERN Colt Scientific Library 1.2.0 [CER], which is also required by JUNG.

Binaries of these libraries are also distributed with the XChange^{EQ} source code in the `lib/` directory.

16.2.3 Building in Eclipse

To build the XChange^{EQ} prototype within the Integrated Development Environment Eclipse [Ecl], it is recommended to use the ANTLR IDE Eclipse plug-in [ANTa].

The ANTLR IDE Eclipse plug-in does not automatically generate the parsers from grammar descriptions as part of the build process of Eclipse. One has to manually select “Generate Code” in each grammar descriptions. In the XChange^{EQ} prototype there are two such grammar descriptions: `src/xchangeeq/compiler/parser/NormalizedXChangeEQ.g` and `src/xchangeeq/minixcerpt/parser/DataTerm.g`.

Other than this, the regular build process of Eclipse should work without any further caveats. You can also use the ANT build file, which is described next, in Eclipse.

16.2.4 Building with ANT

The XChange^{EQ} prototype can also be built using the Apache ANT build tool, version 1.7.0 [Apa]. An appropriate `build.xml` file is distributed with the source code. It requires the ANTLR v3 task for Ant [ANTc], which is also located in the `lib/` directory.

The default target `compile` in `build.xml` simply compiles all classes of the XChange^{EQ} prototype into the directory `bin/`. The additional target `dist` packages all classes into a Java Archive file (JAR) for distribution.

16.3 Overview of Source Code

We now give an overview of the source code of the XChange^{EQ} prototype. The aim is not to cover all aspects but to give a general understanding and point out some salient aspects as necessary to explore the prototype and modify or extend it.

16.3.1 Packages

The source code is subdivided into the following packages and sub-packages:

- Package `xchangeeq` contains the XChange^{EQ} engines that have been described in Section 16.1.
- Package `xchangeeq.commons` contains several classes that are used all over the source code.
- Package `xchangeeq.compiler` contains the classes that are responsible for compiling an XChange^{EQ} program into a physical query plan. They are described in Section 16.3.2.
- Package `xchangeeq.gui` contains some classes to graphically display logical query plans, see Section 16.3.3.
- Package `xchangeeq.logical` contains classes for representing logical query plans, i.e., data structures that are used in the compilation process. The algebra operators are in `xchangeeq.logical.algebra`, conditions for selections in `xchangeeq.logical.conditions`, and query plans themselves in `xchangeeq.logical.queryplan`.

- Package `xchangeeq.minixcerpt` contains a limited implementation of Xcerpt, see also the discussion in Section 16.4.4
- Package `xchangeeq.physical` contains classes for representing physical query plans, i.e., the data structures used in the actual query evaluation. They are shortly described in Section 16.3.4
- Package `xchangeeq.temprlv` defines data structures such as the temporal distance graph that are needed for determining temporal relevance as part of the compilation process.
- Package `xchangeeq.time` contains classes for representing time points and durations.

16.3.2 Compilation Process

Figure 16.3 illustrates the process of compiling a given XChange^{EQ} query program into the physical query plan that is used in all XChange^{EQ} engines for query evaluation. Data structures that are created and passed along in the compilation process are depicted as rectangles with sharp corners. Functions that transform these data structures are depicted as rectangles with rounded corners.

With each data structure and function, the most prominent source file is indicated. Source files for related classes (e.g., the classes for algebra operators that are used in a query plan) are usually located nearby in the package hierarchy.

The functions that transform the data structures are related to the previous chapters as follows:

- *Compilation to logical query plan* (`Ast2Log.java`) implements the translation of rule programs into query plans that has been described in Chapter 14.4 and Chapter 13.4.
- *Finite differencing* (`Log2Diff.java`) has been described in Chapter 14.3.
- *Computation of temporal distances* (`Log2Tdgs.java`) implements the building of the temporal distance graphs as described in Chapter 15.4.1.
- *Determination of temporal relevance* implements the computation of the relevance times and temporal relevance conditions from the temporal distance graphs as described in Chapter 15.4.2, together with their minimization as described in Chapter 15.4.3.

16.3.3 Logical Query Plans

To gain a better understanding of logical query plans in the XChange^{EQ} prototype, we explain some conventions and salient points about them and about and how they are compiled.

Visualization of query plans As an aid for analyzing, testing, and debugging, the XChange^{EQ} prototype includes functionality for graphically displaying logical query plans. Figure 16.4 shows two screenshots of this functionality in the prototype. A query plan is displayed with a tab for each materialization point. In each tab, the corresponding algebra expression for this materialization point is shown as a tree.

Both screen shots show the query plans for the program of Figure 16.1 and have the tab for the materialization point “Q2” and its expression opened The first screenshot (Figure 16.4(a)) shows the query plan after compilation but before finite differencing, the second screenshot shows the query plan after finite differencing (Figure 16.4(b)).

The notation used in the algebra expressions is close to the one that has been used in the previous chapters. Relations on the leaves of the expression tree are followed by the schema that is associated with them, written in square brackets. Note that the screenshots use a triangle symbol $\triangleright$ instead of the symbol $\overline{\ltimes}$ for anti-semi-joins.

It is possible to drag and drop nodes in the expression trees in case the automatic layout does not achieve a satisfying display. When the mouse pointer hovers over a node in the tree, the schema of the output of the corresponding subexpression is displayed as a tooltip.

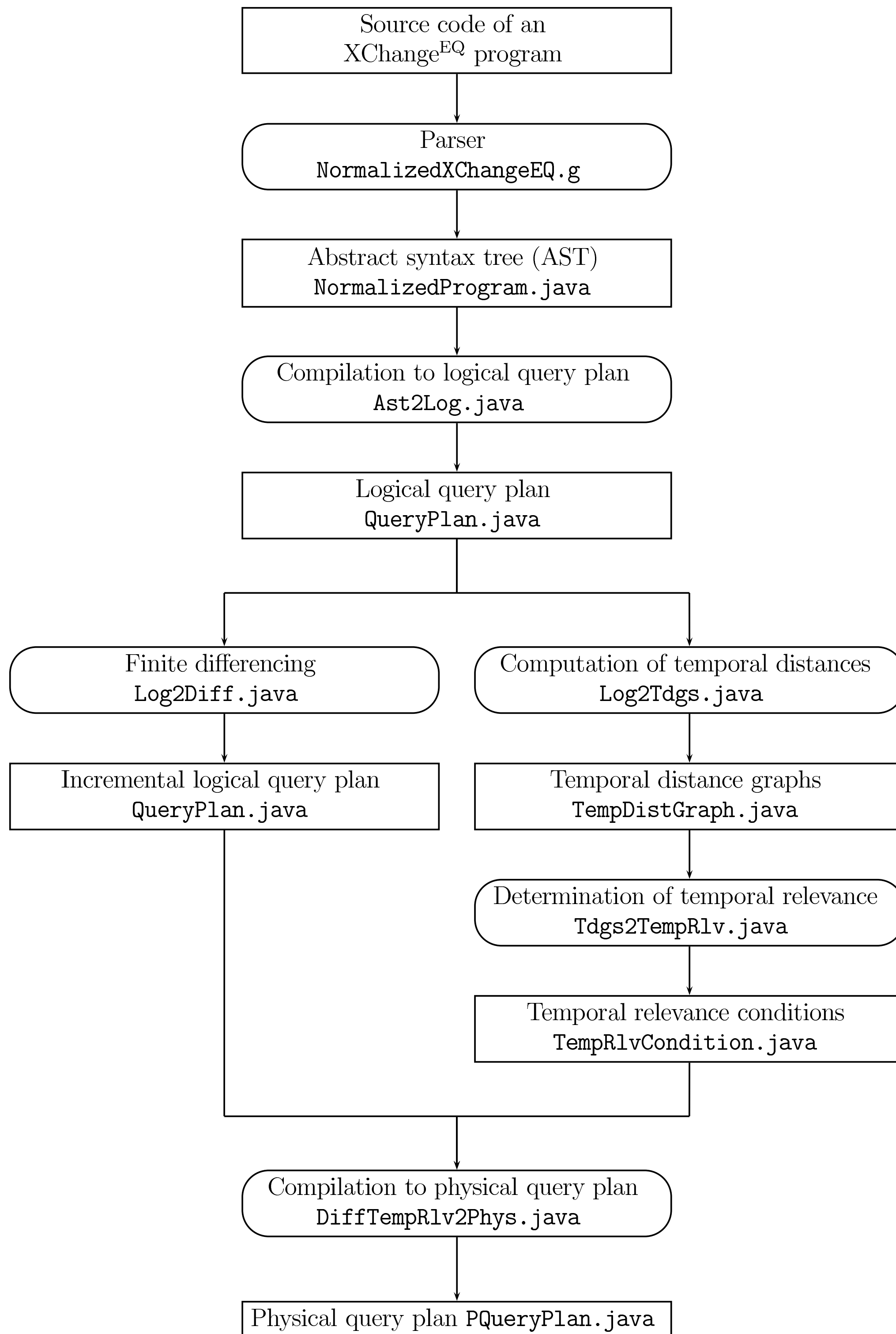
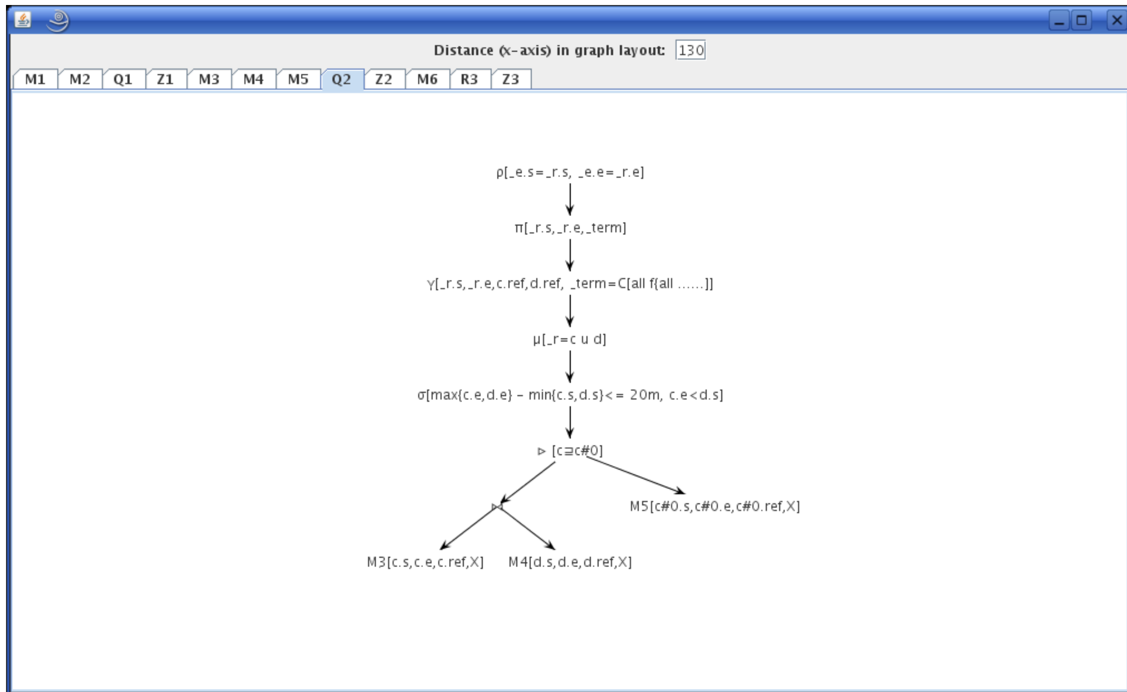
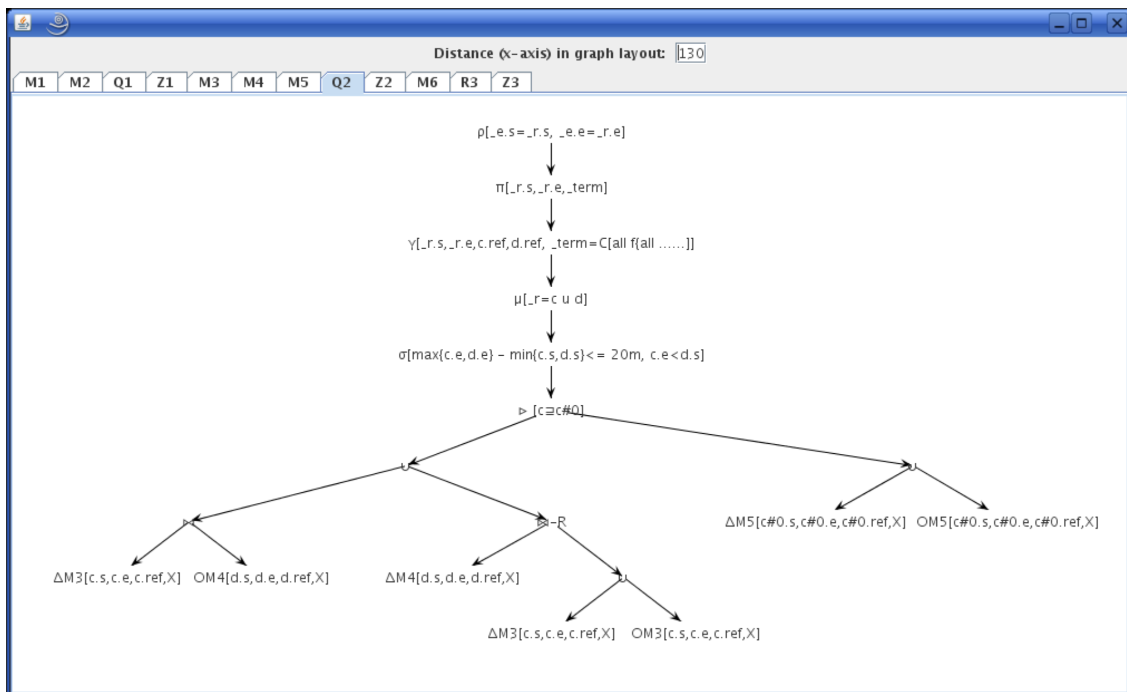


Figure 16.3: Compilation process of an XChange^{EQ} query program into a physical query plan



(a) Before finite differencing



(b) After finite differencing

Figure 16.4: Visualization of logical query plans in XChange^{EQ} prototype

Materialization points In the current XChange^{EQ} prototype, each simple event query has its own materialization point in the logical query plan as described in Chapter 14.4.4. The prototype uses the following conventions for the names of relations and materialization points (n being some integer):

- E is the relation for the incoming event stream.
- Mn are materialization points for simple event queries.
- Qn are materialization points for deductive rules (`DETECT...ON...END`).
- Rn are materialization points for reactive rules (`RAISE...ON...END`).
- Zn are materialization points that collect the results for each stratum.

Schemas Logical query plan use the named perspective on relational algebra. However, all algebra operations in the prototype keep a stable order of the attributes, which simplifies the translation to a physical query plan (which uses the unnamed/positional perspective on relational algebra). The prototype keeps the names of event identifiers and variables from the source code of the XChange^{EQ} program that is compiled.

In the previous chapters, the event identifiers r and e had special significance as the event identifiers for events that are respectively in the result of a deductive rule and in the incoming event stream E. In the prototype, these are named `_r` and `_e` to avoid clashes with a potential use of these event identifiers in rules (i.e., in the form `event e:...`). Event identifiers that are generated due to event accumulation (through `while i:...`, cf. Chapter 13.4.2) are named `i#`n with n some integer.

Finite Differencing In the translation from the abstract syntax tree to a logical query plan, the prototype uses n-ary joins (class `MultiJoin`) rather than just binary joins. Finite differencing of these n-ary joins is done with the second method described in Chapter 14.3.4, which avoids an exponential blow-up in the size of the expression resulting from finite differencing.

It is a common convention for the physical evaluation of a query plan that the smaller relation should be to the left side. The method for finite differencing from Chapter 14.3.4 already orders the relations in a way that delta relations, which can be expected to be smaller than history relations, are to the left side.

Care has to be taken with the ordering of relations in a join however when going from the named perspective on relational algebra to the unnamed perspective because joins are not commutative in the unnamed perspective. An additional reordering of the attributes in the output of a join is needed. To simplify the translation from the logical level (where we use the named perspective) to the physical level (where we use the unnamed perspective), the prototype uses a "reverse join operator," which is displayed as `⋈-R` (e.g., in Figure 16.4). It performs the same operation as a regular join, but orders the attributes in its output as if the arguments of the regular join were swapped.[1]

16.3.4 Query Evaluation

The actual query evaluation is performed with physical query plans that are represented by objects of the class `PQueryPlan` in package `xchangeeq.physical`. Event histories are stored in objects of the class `History`. The method `garbageCollect` in this class performs garbage collection using temporal relevance conditions. Delta relations (i.e., the changes to history relations that are computed in each evaluation step) are represented by objects of the class `Delta`. The class `Delta` has a subclass `ReactionDelta`. Objects of this class are generated by reactive rules (or in turn

[1] As mentioned before, we want to keep to the convention that the smaller relation is on the left side and therefore do not want to directly use a regular join with swapped arguments.

materialization points named Rn in logical query plans) and additionally trigger the reactions specified in the reactive rules.

As the focus of the operational semantics of XChange^{EQ} has been on the logical level, the current prototype does not provide advanced index structures and algorithms for the implementation of relational algebra operations. Joins, for example, are performed as simple nested loop joins.

16.4 Current Limitations of Prototype

The current XChange^{EQ} implementations is a research prototype, not a product. As such it cuts some corners that are not essential for a proof-of-concept. We now describe its current limitations. None of the limitations that will be discussed are inherent to the prototype's architecture, which has been designed to be easily extended.

16.4.1 Explicit Stratification

As pointed out already in Section 16.1.1, the prototype requires that XChange^{EQ} programs indicate a stratification of their rules through `STRATUM ... END` blocks. The reason for this limitation is that the current XChange^{EQ} prototype uses a fairly limited implementation of Xcerpt. This limited Xcerpt implementation supports matching between query terms and data terms (as needed for the matching operator Q^{X} in CERA) but does not support the full simulation unification between query terms and construct terms. For computing the rule dependency graph that is used to stratify programs (cf. Chapter 10.1.2), full simulation unification would be necessary. The algorithm for simulation unification has been details in [Sch04]. The algorithm could, e.g., be implemented as a preprocessor that transforms regular XChange^{EQ} programs into XChange^{EQ} programs that explicitly indicate stratification.

16.4.2 Normalized Programs

The current prototype only supports programs with normalized rules (as described in Chapter 13.4.1). Since any XChange^{EQ} program can be converted into a program that contains only normalized rules, this is not a severe limitation. The prototype can be extended to arbitrary XChange^{EQ} rules easily. Apart from extending the parser accordingly, one only has to implement the rewriting rules of Chapter 13.4.1 as a transformation on the abstract syntax tree so that the compiler can then work only with normalized rules.

16.4.3 Timer Events

Timer events are not supported in the current prototype. They could be added either as base relations (like they were treated in the previous chapters) or by collapsing the base relation and the associated join into one operator. The main issue about timer events is that the query evaluation has to be "woken up" when a timer event happens. For example for a timer event `extend[event i, 1h]`, a query evaluation step has to be performed one hour after each i-event. Scheduling such an event query evaluation step could be realized, e.g., with Java's `Timer` facility (package `java.util`).

16.4.4 Xcerpt Implementation

The current Xcerpt implementation [Xce] has been written in the functional programming language Haskell [Tho99] and there is so far no stable Java implementation of Xcerpt that would have been suited to embed in the XChange^{EQ} prototype. Therefore the XChange^{EQ} prototype provides its own, limited implementation of Xcerpt (in the package `xchangeeq.minixcerpt`). Note that implementing the XChange^{EQ} prototype in Haskell would not have been a good option: incremental event query evaluation is inherently a stateful process that is difficult to express in a functional

programming language (in particular in Haskell which aims at being a “purely functional” language and which uses lazy evaluation).

The Xcerpt implementation in the XChange^{EQ} prototype covers only a limited set of query and construct terms. Supported are currently only the following constructs in them:

- total and partial, ordered and unordered term specifications (`[]`, `[[]]`, `{}`, `{{}}`),
- descendant specifications (`desc`),
- regular variables (`var X`),
- variable restrictions (`var X ->` *term*), and
- grouping with `all` (and an optional `group by`)

Not supported are currently:

- negation of subterms (`without`),
- optional subterms (`optional`),
- label variables (`var X {...}`, etc.),
- position specifications (`pos` n),
- grouping with `some`, and
- functions such as + or `concat` as well as aggregation functions such as `avg` or `max`.

The Xcerpt implementation in the XChange^{EQ} prototype has been designed so that these unsupported constructs can be added easily. The Xcerpt implementation does not focus on efficiency. In particular it does not implement memoization as part of the matching process as described in [Sch04] and [BFLS06].

16.4.5 Actions in Reactive Rules

For reactive rules of the form `RAISE...ON...END`, the current prototype supports only specifying the action of calling a static Java method. Accordingly, the `to(...)` specification that follows `RAISE` (see Chapter 6.5) must contain `transport="java"`. Note that calling a static Java method is the most general case for a reactive rule and all other cases (e.g., console output or calling a Web service) can be programmed using this a static Java method.

16.4.6 Error Reporting and Exception Handling

As a prototype, the current XChange^{EQ} implementation does not pay much attention to good error reporting and exception handling. It does not perform many syntax checks on input XChange^{EQ} programs. When a syntactically incorrect XChange^{EQ} program is encountered, it relies on the error messages that are automatically generated by ANTLR and does not provide further assistance to the user.

16.4.7 Query Rewriting

The current prototype follows a single strategy for introducing materialization points in query plans: all simple event queries have their own materialization points and no further materialization points are introduced. It does not perform any query rewriting that would introduce further materialization points, e.g., to materialize intermediate results that are generated by joins. It also does not perform any query rewriting such as pushing selections that would likely make query plans more efficient.

Such a rewritings could be implemented as transformations of the logical query plan. However, the issue in event querying (and querying in general) is not the rewriting of query plans itself but rather the development of query planners that provide heuristics and cost measures that indicate which of a number of possible query plans obtained through rewriting is more likely to be more efficient (cf. Chapter 14.5).

Part V

Conclusions and Outlook

Chapter 17

Language Design Revisited

Having introduces the syntax, declarative semantics, and incremental evaluation of XChange^{EQ} we now revisit its language design and illustrate some of its advantages over the existing approaches that have been discussed in Chapter 3.

A central design idea in the event query language XChange^{EQ} is to treat each querying dimension separately (Section 17.1), which contributes to high expressivity given that the language is highly expressive in each of the four dimensions event data (Section 17.2), event composition (Section 17.3), temporal relationships (Section 17.4), and event accumulation (Section 17.5). Further we discuss the importance of both deductive and reactive rules in the language design of XChange^{EQ} (Section 17.6), its formal semantics (Section 17.7), and its extensibility (Section 17.8). In a final summary (Section 17.9), we complete the earlier tabular comparison of the existing approaches from Figure 3.7 by including XChange^{EQ}.

17.1 Separation of Concerns w.r.t. Four Dimensions

As mentioned in Chapter 5.1, a sufficiently expressive event query language should cover at least the four querying dimensions event data, event composition, temporal (and other) relationships, and event accumulation. A core idea in the language design of XChange^{EQ} is to enforce a separation of these querying dimensions.

Queries in XChange^{EQ} are written in a style that is reminiscent of logical formulas, and the different dimensions are expressed in different parts of the formula. Event data is reflected in simple event queries and in data conditions of the `where`-clause. Event composition is expressed through the `and` and `or` junctors. Temporal relationships are expressed as temporal conditions in the `where`-clause. Event accumulation finally is expressed through the `while` construct that defines the accumulation window.

Arguably, this separation of concerns yields a clear language design with clear semantics and little potential for misinterpretations. It also makes queries easy to read and understand. More importantly, this separation allows to argue that the language reaches a high degree of expressivity since it is, as we will discuss in the next sections, highly expressive in each individual dimension.

Composition Operators Composition-operator-based languages mix some of the querying dimensions in their syntax and semantics. It can be argued that this impacts ease of use negatively, offers potential for misinterpretations of queries, and can be held responsible for some lacks in expressivity.

The simplest and most intuitive example of mixing of dimensions in composition-operator-based languages is the sequence operator $A;B$. It mixed the composition of events ("A and B must happen") with their temporal relationship ("A is before B"). We now illustrate with example some of the difficulties this causes. In the examples we assume the interpretation of

the sequence operator with non-immediately following events that happen over time intervals (cf. Chapter 3.2.2).

Consider an event query asking for events A, B, C to happen, where A happens before C and B happens before C. Using the temporal conditions in XChange^{EQ}, this is straight-forward:

```
and {
  event a: A,
  event b: B,
  event c: C
}
where { a before c, b before c }
```

With composition operators, one might be tempted to write this event query as $(A;C)\triangle(B;C)$. This however does not yield the intended result since *different* C-events can be used in answering the query. (A correct way to write the query would be $(A\triangle B);C$.) Further examples that show similar potential misinterpretations are in [ZS01, GA02, AC06, BE08b].

As a further example that illustrates a lack of clarity that might make composition operators hard to use, consider now an event query asking for events A, B, C and D to happen, with the constraints that A happens before B, A also happens before C, and C before D. The XChange^{EQ} query is analogous to this natural language description.

```
and {
  event a: A,
  event b: B,
  event c: C,
  event d: D
}
where { a before b, a before c, c before d }
```

Similar to the previous example, note that the query cannot be expressed with composition operators as $(A;B)\triangle(A;C)\triangle(C;D)$ because this would allow different instance of A and C to be used. A correct way to express the query would be $A;(B\triangle(C;D))$.

Consider now, what happens if we only add an additional constraint that B happens before D. In the XChange^{EQ} query we simply have to add this statement as `b before d` to the `where`-clause. With composition operators, however, the new query bears only little resemblance to the old: $A;(B\triangle C);D$. In fact, even though we *added* a constraint in our specification, the number of operators stays the *same*.[1] This can be argued to be quite unnatural and might easily cause programming errors.

For a final example, we consider queries that involve also metric temporal constraints such as "event A and B happen within 1 hour." The following XChange^{EQ} query specifies that A happens, then B happens within 1 hour of that A, and then C happens within 1 hour of that B,

```
and {
  event a: A,
  event b: B,
  event c: C
}
where { a before b, {a,b} within 1 hour,
        b before c, {b,c} within 1 hour }
```

Many composition operator based languages support metric temporal constraints by offering extended operators. For example, $(A;B)_t$ would denote that B happens within t time units after A, and $(A\triangle B)_t$ would denote that A and B happen within t time units (regardless of their order). However with these operators, the above query cannot be expressed. Note that the expression $((A;B)_{1h};C)_{1h}$ would require C to happen within 1 hour of A, not of B (symmetrically for

[1] In composition-operator-based languages that support n-ary versions of the sequence operators (such as `andthen` in [Eck05, BEP06a]), it could even be said to be one operator *less*.

$(A; (B; C)_{1h})_{1h})$.[2] Of course, what can or cannot be expressed in a given event algebra always depends on the operators it offers. For example, an event algebra could also offer a sequence operator of higher arity with temporal constraints and thus be able to express the query (e.g., as $A;_{1h} B;_{1h} C$).

Data Stream Languages Like composition-operator-based languages, data stream languages can also be argued to mix some of the querying dimensions. There, the mixing of dimensions primarily leads to an restricted coverage in each of the dimensions, as will be discussed further down.

Production rules Since production rules are not a dedicated event query language, rather a lower-level programming model that is relatively well-suited for event processing, they offer no built-in support whatsoever for any of the dimensions. As explained in Chapter 3.4), support for certain aspects of event queries such as temporal relationships must be programmed manually, usually by reverting to the host programming language. In this sense, production rules can neither be said to separate nor mix the four querying dimensions — it simply depends on the way programmers design and structure their rules.

It is possible to a large extend to write production rules in a structure that mimics XChange^{EQ} and its separation of dimensions. This particularly so for the first three dimensions, event data, event composition, and temporal relationships. The examples in Chapter 3.4 have been written in such a way so that they are as close as possible to their counterparts in XChange^{EQ}. Mimicking XChange^{EQ} for queries requiring event accumulation is more difficult, primarily because the constructs for it in production rule languages are very low level and not tailored for the temporal aspect of accumulating events rather than facts (cf. Chapter 3.4.4). A separation of the dimensions can however usually still be achieved.

17.2 Event Data and Querying Events in XML formats

Events and their associated data are often represented in XML formats such as SOAP, CBE, or FCML (cf. Chapter 2.4.2). XChange^{EQ} addresses the need to query events in such XML formats by building upon the Web query language Xcerpt [SB04, Sch04].

For embedding it into XChange^{EQ}, Xcerpt has a number of advantages that we believe outweigh its potential disadvantage of not being one of the conventional XML query languages XSLT and XQuery that are standardized by the W3C:

- Xcerpt's query terms describe a pattern for incoming data. As such, they serve a dual purpose in XChange^{EQ} by (1) specifying implicitly a kind of event type and (2) extracting data from events. The (implicit) notion of an event type is relevant in querying events because one typically only wants to compose or accumulate events of certain types. XSLT and XQuery focus primarily on extracting data and are not as well suited for describing also an event type.
- Xcerpt separates access to existing data (with query terms) and construction of new data (with construct terms). This makes it possible to embed Xcerpt's query terms and construct terms as "black boxes" into XChange^{EQ}, while still achieving a language that looks natural and homogenous. Because XSLT and XQuery inter-mix data access and construction [BFB^{+}05], such a "black box" approach for embedding them would be less natural there.
- Xcerpt uses substitutions and substitution sets as output of matching query terms against data and as input to construct terms for constructing new data. This has been a key enable

[2]Note that if we were to take the interpretation of the sequence operator based on time points, then the query could be written as $((A; B)_{1h}; C)1h$. However then the modified query where C is supposed to happen after B within 2 hours of A (as opposed to within 1 hour of B) cannot be expressed.

in the design of Xcerpt as a rule-based Web query language, where rule bodies resemble logic formulas, and has similarly been key enabler in the design of XChange^{EQ} as a rule-based event query language. Further we have seen in this work that this makes declarative semantics with a model-theoretic approach and operational semantics building on relational algebra possible. XSLT and XQuery in contrast are more similar to functional programming languages (esp. in their semantics) than rule-based logic programming languages.

- Xcerpt aims at being a versatile Web query language. Conceptually, it can be used for querying data in different Web data format, e.g., not just XML but also RDF, even within a single query or program. Extending Xcerpt to other Web data formats is an ongoing work, and results obtained there are easily integrated into XChange^{EQ}.
- Finally, Xcerpt has already been successfully built upon in the reactive language XChange, and a design goal of XChange^{EQ} has been that is can be used as an event query language inside the reactive language XChange.

The advantages of Xcerpt aside, XChange^{EQ}'s language design is extensible to build upon other query languages for accessing event data as well. With Rel^{EQ}, which follows XChange^{EQ}'s ideas but represents and queries events as relational facts, we have already seen an example of this (cf. Chapter 6.11). The essential requirement for using another data query language with XChange^{EQ}'s approach to querying event is that it must have the ability to produce sets of variable substitutions when matching against incoming events and the ability to consume sets of variable substitutions for constructing new events.

Composition operators Most composition-operator-based event query languages neglect the aspect of event data (at least as far as their descriptions in the literature and their formal semantics are concerned). They do not build upon an existing query language nor do they define their own query language.

Some composition-operator-based languages support event data in the form of attribute-value pairs (or equivalently relational facts). For querying XML, however, this can be deemed insufficient since XML data might have a more complex structure and require queries that support incomplete specifications (e.g., searching for elements in an unknown depths as supported with XPath's descendant axis or Xcerpt's `desc` construct).

The only composition-operator-based language that natively supports querying XML data that we are aware of are the composition operators of XChange (that have preceded and, through the difficulties found with them, motivated the development of XChange^{EQ}). Like XChange^{EQ}, they build upon Xcerpt.

Production rules Production rules work with the data model of the host programming language (e.g., Java), which is usually object-oriented. Accordingly they usually require to model each type of event with a class and to represent events as objects of these class. There is no dedicated query language for accessing the data of events in production rules; this is done with the constructs of the host programming language.

When events are communicated as XML, this typically requires to serialize and de-serialize between the XML and object representation. Modern development tools offer considerable support for automatic serialization and de-serialization between objects and XML. However, they usually require that the objects (or rather classes) have been defined first and derive from that corresponding XML schemas. The other direction, where the XML schemas for objects are defined first and appropriate classes and objects have to be generated usually involves a significant amount of manual labor by programmers. Further, finding a good object representation for a complicated XML schema (containing, e.g., recursive definitions or many alternatives) is a hard task.

It might be conceivable to access the Document Object Model (DOM, [H$^+$08]) object structure of events in XML formats directly in production rules. However, querying XML through DOM is already very inconvenient if done in regular code and would be even worse if done in production

rules. Therefore, a conversion to an object model as described above is generally preferable despite the involved work.

Data stream languages Data stream languages such as CQL represent events as relational tuples and build upon the existing query language SQL. However the way CQL builds upon SQL is conceptually very different form the way XChange^{EQ} builds upon Xcerpt: CQL converts the event streams received so far into tables that are then operated on by SQL as if they were normal tables, and finally converts the result back. (A comparable approach to deal with XML events would be to create one big XML documents that is the concatenation of all events received so far, then use an XML query language on that document, and finally convert the result document back into events.) XChange^{EQ} in contrast operates "natively" on events and uses Xcerpt only to extract data from events and to construct new event data

Processing XML events with a CQL-like language is possible using the XML extensions to SQL (SQL/XML, [EMK$^+$04]). Arguably, SQL/XML is not the most intuitive way to deal with XML data. SQL/XML essentially allows to query attribute values of a tuple that contain XML fragments and construct new tuples with atomic attribute values (such as strings or integer) from them within a SQL query. (They way SQL/XML works is somewhat reminiscent of the matching operator and construction function in CERA that have been described in Chapter 13.1.6.) When using SQL/XML as part of CQL, this means that in addition to the conversion between event streams and tables there also is a conversion between XML and tables. This can make CQL queries that have to deal with XML events very hard to understand in comparison to similar XChange^{EQ} queries.

17.3 Event Composition and Garbage Collection

Due the separation of the querying dimensions, event composition in XChange^{EQ} is fairly simple and only two operators, conjunction and disjunction, are needed. The operators have clear and intuitive semantics and there is little potential for misinterpretation. The time a complex event is detected (and any reactions executed) is always simply the time that the last constituent event is detected.

Event composition in XChange^{EQ} is sensitive to data: a variable that occurs in different simple event queries of the same complex event query must be bound to the same value. Data-sensitive event composition (sometimes also called "event correlation") is important in practice as evidenced by the queries in Chapter 7.

Event composition entails a need to store events in the query evaluation for some time and thus a need for garbage collection of events that have become irrelevant. XChange^{EQ} supports a sophisticated automatic garbage collection based on temporal relevance.

Composition operators The simplicity of event composition in XChange^{EQ} is in contrast to composition-operator-based languages that provide a multitude of different operators that must be learned by the programmer. The operators do not always have clear and intuitive semantics and have a potential for misinterpretations (e.g., sequence, counting, cf. Chapter 3.2). For most operators the detection time of the complex event is clear, but there are exceptions (e.g., overlaps or counting, cf. Chapter 3.2.5). Some composition-operator-based language further support so-called detection modes that can have an impact on the detection time of a complex event.

Data-sensitive composition of events is often not considered or requires fairly complicated mechanisms such as the local and global keys in Amit [AE04]. Garbage collection is a less involved problem than in XChange^{EQ} because of the limitations on the expressions that can be formed using composition operators (see the discussion in Section 17.1).

Production rules As mentioned earlier (Section 17.1), production rules can partially mimic the ideas of XChange^{EQ}. This especially applies to event composition: like XChange^{EQ}, produc-

tion rules primarily supply a conjunction for this. However, garbage collection must usually be performed manually, which is a considerable inconvenience and potential for programming errors.

Data stream languages Event composition in data stream languages is done by joining the relations that have been obtained from streams with stream-to-relation operators (or "windows"). The composition is sensitive to data. The detection time of a complex event depends on the relation-to-stream operator (together with the evolution of the state of the result relation) and is thus sometimes hard to grasp. Garbage collection is a fairly simple issue since only the stream-to-relation operators determine how long events must be stored. However this entails that programmers have to work on a lower level, describing exactly how long events are stored rather than relying on a mechanism that automatically determines this for them as in XChange^{EQ}.

17.4 Temporal Relationships

XChange^{EQ} offers a very extensive support for expressing temporal relationships in event queries. Because of the separation of the querying dimensions, it is fairly easy to extend the temporal relationships that are currently supported, e.g., also add relationships that rely on domain- or culture-specific calendars.

XChange^{EQ}'s language design also emphasizes a strong distinction between references to time points or intervals that signify timer events and references to time points or intervals that are part of temporal conditions (see Chapter 6.8).

Production rules Production rules offer no built-in support for temporal relationships. They have to be programmed manually as described in Chapter 3.4.2. A distinction between timer events and references to time in conditions is possible, however the distinction is less clear in the syntax (no explicit `where`-clause as in XChange^{EQ}) and its enforcement the responsibility of the programmer.

Data stream languages Some temporal relationships can be expressed in data stream languages indirectly by using temporal windows. For example that events of types A and B should happen within one hour can be expressed by applying a one hour window to stream A and B each. However, this can be considered fairly inconvenient and the temporal relationships that can be expressed are rather limited.

Composition operators Composition operators have extensive support for temporal relationships, however, we have seen in Section 17.1 that there are some (maybe unexpected) limitations in the expressivity that result from the mixing for the querying dimensions.

The distinction between timer events and references to time in temporal conditions is less clear in composition-operator-based languages than it is in XChange^{EQ}. For example in event query such as $A; B; A + 5h$ one might consider $A + 5h$ either as a timer event that is waited for (and thus the detection time is five hours after the A event) or as expressing a temporal conditions (and thus the detection time is that of the B event). For consistency reasons (e.g., with $A; \neg B; A + 5h$) the former should usually be the case. Some languages would express the latter with temporally restricted operators (e.g., $(A; B)_{5h}$); however these allow only to express relative temporal constraints (e.g., within five hours) and do not allow to express restrictions that refer to absolute time points (e.g., before 5pm on a specific date).

17.5 Event Accumulation

Negation and aggregation are treated uniformly in XChange^{EQ} through event accumulation. The reason for this uniform treatment is that both give rise to so-called non-monotonic queries and accordingly require a restriction to some window in querying events. Event accumulation in

XChange^{EQ} is sensitive to data, which can play a role in event accumulation similar to its role in event composition.

XChange^{EQ} offers great flexibility with regards to the temporal windows that can be used for event accumulation: a window is simply specified by another event. This event can be a simple event, a relative timer event, an absolute or periodic timer event, or an event constructed by a deductive rules from other events.

Composition operators Negation is supported in composition-operator-based languages, usually by a ternary operator $A; \neg B; C$ that resembles a sequence. With this operator there is less flexibility with regards to the accumulation window, which is essentially restricted to being given by two events. (Note that composition operators typically have no notion of deductive rules, so that these events cannot be derived events.) As mentioned before, data is an often neglected aspect in composition operators. There is usually no support for aggregation.

Production rules Event accumulation for negation or aggregation is possible in production rules. However the language constructs that are used for this can be considered fairly inconvenient, even outside of the event processing context (i.e., when processing regular facts not events). In an event processing context, there is no direct support for the important event accumulation window that describes a time window over which events are to be accumulated. As with event composition, garbage collection of accumulated events must be performed manually.

Data stream languages Data stream languages support both aggregation and negation. However the latter is often somewhat inconvenient to express, simply because SQL requires the use of a "back door" like `COUNT(X)=0` or `NOT EXISTS`. There is fairly extensive support for temporal accumulation windows, but is can be considered less flexible than the approach of using another (interval-based) event in XChange^{EQ}. In particular, the common case where one event does not happen between two other events ($A; \neg B; C$ in composition operators) is very hard to express in a data stream language. On the other hand, data stream languages allow tuple-based windows for event accumulation, something that is not supported in XChange^{EQ}.

17.6 Support for Deductive and Reactive Rules

Deductive rules are as important for querying events as they are for traditional querying of non-event data. Accordingly, they are an an integral concept in XChange^{EQ}. Beyond their usual use as a reasoning, abstraction, and mediation mechanism, deductive rules in XChange^{EQ} are also important to define events which will be used as accumulation windows in other deductive rules (cf. Section 17.5) or in advanced uses of grouping.[3] XChange^{EQ}'s approach of writing complex event queries in the style of logic formulas is particularly well-suited for the bodies of deductive rules.

The current declarative semantics of XChange^{EQ} restrict the use of recursion to stratified rule programs (see Chapter 10). However, it is conceivable to be less restricted at the price of more involved semantics. Less restrictive approaches have been well-investigated in the non-event context of logic programming and deductive databases and should be applicable to XChange^{EQ} as well. The current operational semantics of XChange^{EQ} are restricted to hierarchical rule programs, but an extension to stratified rule programs is straight-forward (see Chapter 18.1.1).

In addition to deductive rules, XChange^{EQ} supports reactive rules to specify reactions to (complex) events. When XChange^{EQ} is used in conjunction with the reactive Web language XChange, then is possible to express fairly advanced reactions (including, e.g., updates to Web resources) in a homogenous and fairly declarative language.

[3]For example separate deductive rules must be used when event data is aggregated and then events further filtered based on this aggregated value. The reason for this is that XChange^{EQ} like Xcerpt does not have a mechanism like SQL's `HAVING` clause that allows to express conditions on constructed data within the same query. See also Chapter 18.1.3

Composition operators Current languages based on composition operators do not offer any support for deductive rules. Typically composition operators are used as part of reactive Event-Condition-Action (ECA) rules. These reactive rules can be used to simulate deductive rules to some degree, but this has considerable disadvantages that have been discussed in Chapter 5.4.

Production rules Production rules are reactive rules, although they are often used (also in other contexts than event processing) to express deductive knowledge. The fact (or event) that is derived from some condition over facts (or events) is simply asserted in the action part of a rule. Many production rules languages support the notion of a "logical assertion," which has the effect that a fact (or event) is automatically retracted when its associated condition does not hold anymore. In this respect, simulating deductive rules with production rules is slightly better than simulating deductive rules with the ECA rules that are associated with composition operators. However production rules still offer no support for true deductive rules.

Data stream languages Many data stream languages support views, which can be considered a form of deductive rules. However they are typically more limited than the deductive rules in XChange^{EQ} and do not permit recursion or permit only linear recursion. Data stream languages typically have no notion of reactive rules. Reactions to complex events are typically implemented separately by some entity that consumes output streams of the data stream management systems.[4]

17.7 Formal Semantics

With its model theory and accompanying fixpoint theory, XChange^{EQ} provides highly declarative semantics for querying events based on an approach that is well-established for traditional, non-event queries. Consideration of event data is inherent to the semantics of XChange^{EQ}. Since these semantics work directly on streams and are stateless, they are also very intuitive. A possible drawback of stateless semantics is however that XChange^{EQ} does not support event instance consumption and selection (as found in composition-operator-based languages) or tuple-based windows (as found in data stream languages).

Composition operators Many composition-operator-based languages neglect formal semantics, in particular as far as event data is concerned. When selection or consumption should be described formally for composition operators, the semantics necessarily become stateful, losing a declarative nature and becoming harder to understand.

Production rules The semantics of production rules can be considered intricate and undeclarative. They are stateful and have much in common with the semantics of imperative programming languages. Conflict resolution between rules complicates semantics of production rules further. Because production rules are not a dedicated event query language, their semantics do not work directly on event streams but rather depend on a conversion of events into facts, which might even happen externally to the rules.

Data stream languages The semantics of data stream languages can be said to not work directly on event streams, because they essentially require the conversion of the event streams into relations (through the stream-to-relation operator). The conversion back to streams makes the semantics also stateful: relation-to-stream operators such as `Istream` and `Dstream` depend on the difference between the current result of a SQL query and its previous result.

[4]Note however, that many data stream management systems are engineered in a way that this "separate entity" can run in the same operating system process as the data stream management system. This is important in some applications that require low latency (such as algorithmic trading) to avoid the delay caused by a context switch between different processes.

17.8 Extensibility

The language design of XChange^{EQ} emphasizes extensibility. Its extensibility is greatly aided by the separation of dimensions. Examples of where XChange^{EQ} might be extended fairly easily include the following:

- further temporal relationships, e.g., based on culture- or domain-specific calendars,
- further relative, absolute, or periodic timer events, again e.g, based on culture- or domain-specific calendars [BRS05],
- support of other relationships between events such as causal [Luc02] or spatial relationships [Sch08],
- extensions to the underlying Web query language Xcerpt such as support for querying RDF, which are currently investigated [BFLP08],
- embedding another query language for querying simple events, which might also be for events in other data formats than XML,
- access to non-event data (e.g., in a database or in Web documents) in event queries, and
- user-defined functions for constructing new event data and for testing conditions on event data.

Note that the first two points, temporal relationships and timer events, have potentially an impact on temporal relevance, which has been discussed in Chapter 15.6.2.

Composition operators Extending a composition-operator-based language typically involves defining new operators. This is can be considered more invasive into the language and its design than the possible extensions to XChange^{EQ} that have just been mentioned.

Production rules Because of their intimate connection with a host programming language, production rules can be extended fairly easy in most directions. The price however is that production rules work on a relatively low abstraction level.

Data stream languages Some data stream languages support user-defined functions and windows. However, other issues such as adding new relationships between events or embedding a query language for querying simple events can be considered hard and fairly invasive to the language.

17.9 Summary

Figure 17.1 summarizes how XChange^{EQ} relates to the other approaches for querying events. It completes the table that has previously been given in Figure 3.7 of Chapter 3.5. We refer to the previous sections in this chapter and to Chapter 3.5 for a deeper discussion of the individual points in the table.

	Comp. op.	Data stream lang.	Production rules	XChange^{EQ}
Temporal aspects	++	0	−	++
Negation	+	0	−	++ (data-sensitive, flexible windows)
Aggregation	−−	++	−	++ (flexible temporal windows)
Consumption and selection	++	−−	0	−− (not supported)
Facts and States	−	+	++	−/+ (currently not supported, but extensible)
Formal Semantics	0	+	−	++ (intuitive, highly declarative)
Ease-of-Use, Learning Curve	+	0	−	++
Occurrence and detection time	+	−	−−	++
Extensibility, flexibility	−	+	++	++
Data model: XML support	−−	+	−	++ (builds on Xcerpt)
Integration	see Fig. 3.7	see Fig. 3.7	see Fig. 3.7	standalone, reactive Web language (XChange), prog. lang. (Java)
Implementations	0	++	+	−/+ (only prototype, but operational semantics like data stream lang.)
Development tools	−	−	0	−

Figure 17.1: Summary of the comparison between XChange^{EQ} and other approaches for querying events; completes summary of Figure 3.7 with an additional column for XChange^{EQ}. See Chapter 3 for further details and references to the considered languages.

Chapter 18

Future Work on XChange^{EQ}

This thesis has put forward the foundations of the event query language XChange^{EQ}. However the work also opens up opportunities for future work in the realm of querying events. In this chapter, we discuss concrete future work on the language XChange; the next chapter will broaden scope to issues in querying events and beyond that are not tied directly to XChange^{EQ}.

18.1 Rules and Language Design

We start by discussing future work on XChange^{EQ} that relates to the current design of XChange^{EQ} and the use of rules in it.

18.1.1 Stratified Programs and Beyond

The operational semantics that were presented in this work (Chapters 12–16) and the current prototype implementation focus on hierarchical rule programs, that is, programs without any recursion cycles. The operational semantics and their implementation can easily be extended to stratified programs like they are covered by the declarative semantics of this work (cf. Chapter 10).

To this end, the notion of query plans would have to be extended to allow cyclic definitions of materialization points (with restrictions that correspond to stratification). When an evaluation step is initiated in the incremental evaluation, it would not just preform the computation of changes to event histories once but repeat the process of computing changes and adding these changes to the histories until there are no further changes (i.e., until a fixpoint is reached). Note that this corresponds closely to the fixpoint theory of the declarative semantics and also the semi-naive evaluation of datalog programs [AHV95].

It might also be interesting to develop both declarative and operational semantics for XChange^{EQ} that go beyond stratified programs. An example might be well-founded semantics that are popular for traditional non-event rule languages. It is likely that approaches that are currently investigated for Xcerpt and its non-standard, asymmetric unification [Est08] will be highly relevant to XChange^{EQ}. How important non-stratified rule programs are in practice for event queries, can be considered an issue that is still unresolved.

18.1.2 Modularity

When developing large event query programs, it becomes important to be able to subdivide these programs into modules. Such modules might group together rules that logically belong together and limit the interaction with rules in other modules. At present, XChange^{EQ} offers no module system. However, there is an approach for a module system for Xcerpt [ABB+07] that would also be applicable to XChange^{EQ}. This approach relies on a source-to-source transformation

that rewrites modular rule programs to regular rule programs. To implement this approach for XChange^{EQ}, therefore, no additional work on declarative and operational semantics is necessary.

18.1.3 Syntactic Sugar

The current language design of XChange^{EQ} (and the underlying Web query language Xcerpt) focuses on providing a core of language constructs but avoids introducing unnecessary "syntactic sugar." Syntactic sugar here means language constructs that do not increase expressivity but that might be convenient to make some common expressions easier and shorter to read and write.

As an example of syntactic sugar that might be convenient in XChange^{EQ} consider a query that some event B does not happen between event A and C. This query, which can be expressed very compactly in composition-operator-based languages (as $A; \neg B; C$), becomes somewhat longwinded in XChange^{EQ}, because it requires the use of an auxiliary rule to generate the event accumulation window:

```
DETECT
  AC
ON
  and {
    event a: A,
    event c: C
  }
  where { a before c }
END
```

```
DETECT
  R
ON
  and {
    event w: AC
    while w: not B
  }
END
```

By introducing syntactic sugar in the form of a `from-until` construct as supplement to the event accumulation with `while`, one might write such a query more compactly as something like:

```
DETECT
  AC
ON
  and {
    event a: A,
    event c: C,
    from a until b: not B
  }
END
```

A similar example where auxiliary rules are needed is when event data is to be aggregated and a certain complex event to be generated only if some conditions hold on this aggregated value. This is not uncommon in event queries, especially if cleaning of sensor data is involved (cf. the use case in Chapter 7.2). For a simple example consider the following rule program that sums up the prices of items in an order and then detects orders over a given threshold:

```
DETECT
  ordersum[ var ID, sum(all var P) ]
ON
  event o: order {{
    id { var ID }
    items {{
      item {{
        price { var P }
      }}
    }}
  }}
END
```

```
DETECT
  bigorder [ var ID ]
ON
  event o: odersum[var ID, var S]
  where { var S > 1000 }
END
```

By introducing an additional `having`-clause (in allusion to `HAVING` in SQL) that can operate on aggregated values this might be written more compactly as a single rule:[1]

[1] Note that such a `having`-clause would be as desirable in the query language Xcerpt as it is in XChange^{EQ}.

```
DETECT
  bigorder [ var ID ]
ON
  event o: order {{
    id { var ID }
    items {{
      item {{
        price { var P }
      }}
    }}
  }}
  having { sum(all var S) > 1000 }
END
```

Note that avoiding auxiliary rules by syntactic sugar as in these two example may not just be a matter of convenience. In practice limiting the use of rule chaining can also help avoiding programming mistakes that are due to unindented interaction of rules. Further the syntactic sugar may simplify the identification of optimization opportunities for query evaluation. For example if the condition on the sum above were "$\leq$ 1000" instead of "> 1000," one could stop evaluation as soon as the partial sum so far exceeds 1000.

18.2 Time

XChange^{EQ} emphasizes the important role time plays in querying complex events and is highly flexible and extensible in this respect. We now discuss some opportunities for future work that relate to time in XChange^{EQ}.

18.2.1 Time Model

The current semantics of XChange^{EQ} use a simple linear notion of time with a single time axis. As discussed in Chapter 2.4.4, this notion of time may not be the right choice for some distributed applications. Such distributed applications may require multiple time axes, where each axis models time given according to a different local clock in a distributed systems, or they may require a time domain that is only partially ordered. Both the declarative and operational semantics of XChange^{EQ} have been designed with that in mind and the current time model can be exchanged. In particular, modeling time stamps as attributes of event tuples in the operational semantics helps greatly. We have, for example, already discussed the effect multiple time axis have on temporal relevance of events in Chapter 15.6.1. To conduct such investigations in XChange^{EQ} it may be particularly important to derive the motivation for a different time model from a practical application.

18.2.2 Advanced Calendric Specifications

Even though XChange^{EQ} offers already more extensive support for timer events and temporal relationships than other event query language, some applications may still not be satisfied with the current constructs. In particular, in many business applications domain- or culture-specific concepts such as holidays or work hours are relevant [BRS05]. These concepts may have complicated definitions (e.g., the date of Easter), be subject to irregularities or arbitrary definitions (e.g., no regular date for company outing), and require different treatment in different context (e.g., date of company outing counted as working day in payroll accounting but not for handling of customer orders).

In recognition of the complicated nature of such time- and calendar-related notions, XChange^{EQ} has been designed to be very easy to extend in its support for timer events and temporal relationships. Since it seems impossible to arrive at a set of core constructs that can be considered sufficient for all applications, it may be particularly relevant to investigate how such calendars can be defined externally from XChange^{EQ}. Such external definitions may take the form of simple

tables that represent calendars (up to a specific date), functions written in some regular programming language, or integration of external temporal reasoners. While XChange^{EQ} is open to all these solutions it remains to explore issues such as the design of an Application Programming Interface (API) for such calendric definitions and the impact of such calendric definitions of garbage collection through temporal relevance (cf. also Chapter 15.6.2).

18.2.3 Occurrence Time of Complex Events

In XChange^{EQ}, the occurrence time of a complex event, i.e., an event that is derived by a deductive rule, is the time interval covering all its constituent events, i.e., all events that have been used for answering the query in the rule body. This has the advantage of being a consistent and natural definition.

There may be some applications where it is more natural to assign another occurrence time to the complex event. For example it is conceivable that an alarm that is generated as a combination of sensor events should be given the time point of the last sensor event not the time interval of all sensor events. This is particularly important when this alarm is further queries in other rules. An extension may be made to XChange^{EQ} to allow programmers a choice for the occurrence time of complex events. Note however that this may affect the suitability of the semantics of XChange^{EQ} for event streams (cf. Chapter 11.2).

Further there may be applications that want to distinguish the occurrence time and the detection time of complex events. For example, the time when a fire outbreak will be detected (through sensors, human reports, etc.) will generally be later than its actual occurrence. Note that similar issues exist in the research area of temporal databases, where it is common to distinguish for example valid time and transaction time [JCG^{+}92].

18.2.4 Boundaries in Event Accumulation

Event accumulation over a window w with the keyword `while` in XChange^{EQ} accumulates all events with an occurrence time t such that $start(w) \leq start(t)$ and $end(t) \leq end(w)$. We have seen in Chapter 7.2.4 a case where we would rather like to accumulate events with $start(w) \leq start(t)$ and $end(t) < end(w)$, i.e., with one of the equalities being strict. Whether the inequality should be strict or not matters in particular in cases where the event that (indirectly) defines the accumulation window is of the same type as the events that are accumulated. As an example consider the following:

```
DETECT
  b { count(all var X) }
ON
  event a: a{{ }},
  event w: timer:extend-backward[event a, 1 hour],
  while w: collect a {{ var X }}
END
```

With a strict equality ($end(e) < end(w)$) the values of the a event will not be counted in the construction of the rule head. However with a non-strict equality ($end(e) \leq end(w)$), they will be counted. Clearly it important to support both options in practice.

We might do this in XChange^{EQ} by adding variants of the `while` keyword where the boundaries of the event accumulation window are excluded from the accumulation. In total, this would lead to four variants of `while`: The regular `while` (or `while(incl,incl)`) where both boundaries are included, the variant where the left boundary is included and the right excluded (`while(incl,excl)`), the variant where the left boundary is excluded and the right included (`while(excl,incl)`), and the variant where both boundaries are excluded (`while(excl,excl)`).

For the declarative and operational semantics such variants of `while` do not pose any challenges. It might however be a challenge to find a more intuitive syntax than the suggestion above and also several variants of `while` might be a challenge to programmers.

An alternative to variants of `while` would be to support relative timer events that are not always closed time intervals (such as the current ones) but that may be left-open, right-open, or open on both ends. Because having open time intervals will affect the semantics of other language features (such as temporal relationships) and are probably not easy to understand for programmers, variants of `while` seem preferable though.

18.3 Data and State

The next topics for future work that we discuss relate to processing of data in event queries. When data may change during complex event detection this immediately leads to issues related to state.

18.3.1 Support for Self-Definable Functions

The event query language XChange^{EQ} and its underlying Web query language Xcerpt offer several predefined functions such as addition or string concatenation for computing new data that can be used in conditions or in construct terms. However, at present neither XChange^{EQ} nor Xcerpt offer capabilities for adding and using self-definable (or user-definable) functions, that is, functions that are defined and programmed by the user not the language itself.

Both querying of XML data and querying of events call for self-definable functions. Such self-definable functions may for example be used to express specific financial calculations (e.g., when processing market data) or to apply certain averaging or smoothing algorithms to measured data (e.g., for cleaning of sensor data).

Primarily, support for self-definable functions is an issue that resides with Xcerpt and not so much with XChange^{EQ}, which simply "inherits" the capabilities of Xcerpt in this respect. However, the incremental manner in which event queries are evaluated may make API design for self-definable functions and their incorporation into the operational semantics an interesting issue for XChange^{EQ}: for some functions, especially aggregation functions that work with data that is collected over time, we may want to provide an API that also allows our self-defined functions to compute their results incrementally.

To keep with the declarative nature of the query languages, it will usually be desirable that self-defined functions are true functions, that is, if called with the same arguments they will always deliver the same results and will not cause any side-effects.

18.3.2 Combined Access to Event and Non-Event Data

At present XChange^{EQ} operates only on event data; it does not support accessing non-event data, e.g., of database tables or regular Web documents, in its queries. In practice, combined access to event and non-event data can be important, however. For example sensor events in a Supervisory Control and Data Acquisition (SCADA) application might contain only simple identifiers for the sensors. However some query might require further data about the sensor (e.g., its location), which is stored in some database. As a further example, order events in an order processing application might contain only the customer number, which then must be used to look up further data (e.g., name and address of the customer) in a database.

When non-event data is completely static, i.e., does not change over time, realizing combined access to event and non-event data is a fairly simple issue. In fact it might be realized in a manner similar to self-definable functions (see above).

When non-event data is however dynamic so that is changes over time, possibly during the course of the detection of a complex event, this raises an important semantic question: Which of the various states the non-event data takes over time should be used for answering the complex event query?

A possibility to integrate access to dynamic non-event data in XChange^{EQ} might be to specify in queries the time point that we want to access non-event data. To this end, we could introduce literals of the form `at i:` *q* (in addition to `event i:` *q* and `while i:` *q* literals). These literals

would specify to perform a non-event query q (i.e., a regular Xcerpt query to some Web document) at a time point that is determined by some event i. For example, the following rule would detect a customer's order as combination of an offer and its acceptance (cf. Chapter 7.1.1), using the shipping address from a Web document `http://example.com/customers.xml`. In our query, this Web document is queried when the offer is issued, not when it is accepted. If the customer address is changed after the offer, the result of the query is not affected.

```
DETECT
  order {
    id { var I },
    customer { var C },
    product { var P},
    quantity { var Q },
    discount { var D },
    ship-to { var S}
  }
ON
  and {
    event o: weekly-offer {
      code { var K },
      product { var P },
      discount { var D },
      max-quantity { var MQ }
    },
    event a: accept-offer {
      code { var K },
      id { var I },
      customer { var C },
      quantity { var Q }
    }
    at o: in { resource {"http://example.com/customers.xml"}
      customers {{
         customer {
           name { var C },
           ship-to { var S }
         }
      }}
  } where { o before a, {o,a} within 1 week, var Q <= var MQ }
END
```

Integrating such an access to non-event data in the declarative semantics of XChange^{EQ} would entail adding a set that captures the non-event data to interpretations. That is, we need a set I' of non-event facts (associated with a time interval over which they are valid) in addition to the set I of events that "happen" (cf. Chapter 9.3).

Since they are based on relational algebra, the operational semantics of XChange^{EQ} are well-suited for integrating access to non-event data. However there are some important design issues and research questions for an incremental evaluation. For example, will the event query evaluation engine be notified when non-event data is updated? Does the engine have to ensure that non-event data is accessed at the right time or can it issue historical queries against the non-event data? Note also that query optimization and efficient query evaluation in such scenarios have been given only little consideration in research so far.

18.3.3 State-Based Processing

Like composition operators and data stream languages, XChange^{EQ} focuses on querying complex events, that is, combinations of events occurring over time and has little support for performing state-based processing, which is a particularly strong point of production rules. State-based processing can be important in some event processing applications and we have discussed a case of this in Chapter 3.4.5 with the example of reacting to certain thresholds on the number of persons in a room (which increases and decreases according to events such as "enter" or "leave").

It may be conceivable to extend the event query language XChange^{EQ} in way that it also includes constructs for dealing with this kind of state-based processing. For example, with an extension to querying non-event data (see above) one might also try to add production rules to

XChange^{EQ}. Such production rules would react to changes in the state of non-event data rather than to incoming event messages.

With the integration of production rules (or other reactive rules), giving declarative semantics for the event query language, however, becomes very hard or even impossible. It may therefore also be interesting to investigate alternative approaches that allow only more limited state-based processing but in a more declarative and less imperative manner than production rules.

18.3.4 Access to Time and Event References as Data

XChange^{EQ} makes a fairly strong distinction in its language design between explicit regular data in events and implicit "meta-data" about events such as occurrence time. There may be cases, where it is desirable to cross the borders between time and data more easily.

For example, when constructing a new derived event, we might want to include a string representation of its occurrence time or the occurrence time of some constituent event as a data attribute. This is particularly relevant when the derived event should be processed by some other entity than XChange^{EQ}.

In a similar manner, events might contain in their data string representations of times, dates, or durations that should be used in temporal conditions. For example, a special offer event might contain a data attribute that indicates how long this special offer is valid. When an acceptance event to this special offer is received, we might to check whether offer and acceptance are within the valid time (which is in event data, not a fixed temporal duration that is specified in the query as in the example of Chapter 7.1.1).

Because event data and time stamps are treated uniformly in operational semantics, interchanging data and time stamps is fairly easy to realize there. Determination of the temporal relevance may require more dynamic approaches however, e.g., in the second example the relevance time of special offer events would be affected by a valid time in those events and, since it is part of data, this valid time is not known at query compile time.

A related aspect is that in our operational semantics we generate unique identifiers (called event references) for events with the matching operator. It may be worthwhile to provide capabilities to access these event references also on the level of the query language. In scenarios like order processing, we often have to generate a new order number (cf. Chapter 7.1.1). This order number must be unique so that the event reference (or a number that is computed from it, e.g., using a self-definable function) could be used for it.

18.4 Event Query Evaluation

The operational semantics in this thesis focus on the logical level of an efficient incremental evaluation of event queries. Much remains to be done here, however, to obtain an efficient event query evaluation engine, especially on the physical level. We focus here mainly on issues that are closely related to XChange^{EQ} and its logical query plans. Event query evaluation, also on the physical level, will be discussed further in the research perspectives in the next chapter (Chapter 19.2).

18.4.1 Arrival Order of Simple Events

The current operational semantics assume that events arrive in the order of their occurrence times (or, more precisely, the end time stamps of their occurrence times). This may not always be the case, e.g., due to different delays in communication of a distributed system. Typically there are however still some limits on the order of events, e.g., a maximal delay for events, which is also often called scramble bound.

There are various possibilities to extend the current operational semantics to deal with events that arrive in such an unordered manner. The simplest solution would be to just buffer events for a time determined by the scramble bound and reorder them before feeding them into the evaluation engine. While this does not affect the throughput of query evaluation (i.e., volume of event data

that can be processed in a given time unit), however, it increases latency (i.e., the delay between the last constituent event and detection of a complex event).

An alternative that does not affect latency in this way may be to allow the query evaluation engine to deliver unordered results, that is the order in which complex events are detected may not correspond to their occurrence times. For monotonic queries, that is queries that do not involve negation or aggregation, this is easily done with the current operational semantics. For non-monotonic queries, it is in general not possible to avoid a latency up to the order of the scramble bound unless we accept that the event query evaluation may deliver incorrect results.

An option may be to accept that the query evaluation engine may produce incorrect results for non-monotonic queries at first, but require that it will produce corrections later. This might be useful in some applications where it is preferable to raise (potential) alarms early and, if they turn out to be false alarms, give an all-clear signal later.[2] To incorporate such corrections this into the incremental evaluation of Chapter 14, it would be necessary to not only compute positive changes $\triangle R$ but also negative changes $\triangledown R$.

18.4.2 Query Rewriting and Query Planning in XChange^{EQ}

In this work, we have introduced logical query plans for XChange^{EQ} and discussed that these query plans are well-suited for performing logical query optimization by rewriting of relational algebra expressions. In practice, this requires the development of a query planner component that uses heuristics and cost estimations to transform and select query plans. The development of query planners and their prerequisites such as cost estimation functions is a general topic for research in querying complex event that has not received very much attention so far. We therefore will discuss it in the next chapter. However, there are also some issues that are specific to the logical optimization of XChange^{EQ} because of its nature as a rule language or because it builds upon Xcerpt. These will be discussed next.

18.4.3 Avoiding Intermediate Construction

In the current query plans, matching and construction are black box operators. When rule chaining is used, we always first apply a construction and then match against the constructed result. This intermediate construction can be a source of inefficiency in cases where we first "wrap" some data in the construction (by putting it into XML elements) only to "unwrap" it again in the following matching (by extracting it from the XML elements). As an example consider the following two rules, where the second rules queries the results of the first:

```
DETECT
  c{ b{var X} }
ON
  event a: a{ var X}
END
```

```
DETECT
  d{ var X }
ON
  event c: c{ b {var X} }
END
```

These two rules may be more efficiently written in just one rule that avoids wrapping and unwrapping the values for variable X:[3]

```
DETECT
  d{ var X }
ON
  event a: a{ var X}
END
```

[2] Note that in a concrete application involving human recipients of alarms (e.g., in health care), there is a trade-off between raising alarms early and generating too many false. Many false alarms may lead to alarm fatigue that tempts human recipients to ignore or react slowly to some alarms.

[3] This transformation is not fully correct: the stream of incoming events might contain events that match the query from the second rule `c{b{var X}}`. However, we ignore this here to keep things simple.

Note that such an optimization may be done either on the level of logical query plans or as a source-to-source transformation. In either case, it requires an intimate knowledge about simulation unification and essentially means reasoning about query equivalences. A further variant of such optimizations is that there may also be schema constraints on events.

18.4.4 Goal-Directed Forward-Chaining

Another potential source of inefficiency in our present operational semantics for XChange^{EQ} is that their forward-chaining evaluation is not goal-directed. Accordingly it may compute results that are not actually needed. Consider as a trivial example the following program containing one deductive rules and one reactive rule.

```
DETECT
  b { var X }
ON
  event a: a{ var X }
END
```

```
RAISE
  to(...) {
    d{ var X}
  }
ON
  event c: c{ var X }
  where { var X > 100 }
END
```

Eventually we only care about the reactions such a query program causes; the results of deductive rules are not relevant outside of our query program. In this sense reactive rules are the counterpart of goals in logic programming and deductive databases. Since the deductive rule in our example derives events that are not relevant to any reactive rule in the program, it may be dropped and the reactions would stay the same.

A more involved example, where a deductive rules may generate events that are not relevant to any reactive rule, is the following.

```
DETECT
  b[ var X, var Y ]
ON
  event a: a[ var X, var Y ]
END
```

```
RAISE
  to(...) {
    d[ var X ]
  }
ON
  event c: b[ var X, "const" ]
END
```

Here, the reactive rule is only interested in events that match `b[var X, "const"]`, i.e., where the second subterm has the constant value `"const"`. However, the deductive rule will also derive other events (e.g., an event `b["1","2"]` when there is an input event `a["1","2"]`) that are not relevant to the reactive rule. To avoid production of such irrelevant events, the deductive rule might be rewritten so that the variable `var Y` is replaced with the constant value `"const"`.

Such source-to-source transformations to make the evaluation of rule programs more efficient in a forward-chaining manner by making it goal-directed have been investigated in depth in the context of logic programming and deductive databases. (Note that such rewritings are more complicated than the two illustrated above as soon as the programs contain recursion cycles.) The most popular technique is called magic set rewriting (see, e.g., [AHV95]). In principle, such techniques are also applicable to XChange^{EQ} (and a forward-chaining evaluation of Xcerpt). The main issue in transferring them would be to deal with the non-standard unification between query terms and construct terms. In the Xcerpt context, this issue is currently being investigated (see [BEFL08] for first results).

18.4.5 Investigation of Backward-Chaining

Rewriting techniques such as magic set essentially aim at simulating the effect of a backward-chaining (or top-down) evaluation in a forward-chaining evaluation. In particular, they simulate

the pushing of selections[4] that is done naturally in a backward-chaining evaluation and (ideally) produce only those facts (or events) that are relevant to a goal (or reactive rule) in the program [AHV95]. We refer to [AHV95] and [Bry90] for more details on the relationship between these rewriting techniques and backward-chaining.

Naturally, this leads to the question if backward-chaining algorithms could be used for the incremental evaluation of event query programs with deductive rules as well and how. A simple SLD-resolution [Llo93, AHV95] can be argued to be not suitable since it will not store intermediate results as desirable for an incremental evaluation: it is essentially a depth-first search of the proof tree and intermediate results that are not on the path from the current node to the root are not stored.

However, more advanced backward-chaining algorithms such as QSQ [Vie86, AHV95], OLDT resolution [TS86], and related approaches employ memoing (also called tabling) [War92], that is, they store intermediate results. In principle therefore, these algorithms may be extended to work incrementally in the sense that new data arrives during query evaluation. Note that while these algorithms assume that all extensional data is finite and readily available when the query evaluation starts, they already are in a sense incremental because query evaluation continually derives new intensional data with deductive rules. The major difference is that query evaluation has no control over what data may arrive next in an event stream but has some control over which data is derived next because it also controls the order in which deductive rule are evaluated.

The issues in applying a backward-chaining approach to an incremental event query evaluation can be summarized as follows. During query evaluation new *extensional* data arrives in the event stream in an "uncontrolled" manner. Data structures that explicitly or implicitly represent proof trees must be kept in memory and this may be a substantial overhead (compared to the query plans in a forward-chaining evaluation). Garbage collection of events that become irrelevant due to the progression of time is necessary. In the context of backward-chaining algorithms, garbage collection may affect not only stored ("memoized") incoming events and intermediate results but also involve removing subtrees of the proof tree (or other parts of similar auxiliary data structures). Finally, a strength of the forward-chaining method presented in this work is that is allows different materialization strategies, that is, it gives a fine control over which intermediate results are stored and which must be recomputed across evaluation steps. Ideally, a similarly fine control should also be given with a backward-chaining algorithm.

18.4.6 Efficient Implementation, Experimental Evaluation

The current XChange^{EQ} prototype (cf. Chapter 16) focuses on the logical level of query evaluation. This is in line with the focus of the operational semantics given in this thesis. While the prototype provides a physical level for the actual query evaluation, it does not pay much attention to using efficient index structures and algorithms on the physical level. In particular it performs joins as simple nested loop joins rather than using more advanced algorithms such as merge or hash joins that would also require appropriate index structures. The prototype also does not consider issues such as efficient management of memory buffers, operator scheduling, or the use of an abstract machine for query evaluation.

There has been quite some work on efficient evaluation of event queries (and other streaming queries) on the physical level, especially in the context of data stream management systems. Since relational algebra is also the basis of these systems, the ideas and results from there are in principle applicable for evaluating XChange^{EQ}. It may in fact also be considered to translate from the logical XChange^{EQ} query plans into the physical query plans of some data stream management systems.[5]

Given this previous work and the amount of work that would have to go into building a really efficient physical evaluation for XChange^{EQ}, a conscious decision has been made for the research

[4] In the second example from Section 18.4.4 above, the "selection" is that variable Y must be bound to value `"const"`.

[5] In this case, XML matching and construction would have to be integrated into these systems. Further it may be advisable to use a data stream management system that supports predicate windows (cf. Chapter 3.3.2), since the temporal relevance conditions of XChange^{EQ} are more complicated than simple sliding windows.

of this thesis to focus on the logical level of XChange^{EQ}. This decision has also been motivated by the much-discussed lack of formal foundations in querying events. In particular, work on data stream management systems so far is almost exclusively concerned with the physical level of query evaluation and the logical level as well as issues such as language design, declarative semantics, and correctness of operational semantics are not discussed. Rather than adding to the work on the physical level (e.g., by concentrating on particular classes of queries or types of event data distributions), this thesis seeks to address the lack of formal foundations.

Since we have not focused on physical level in the current XChange^{EQ} prototype, experimental measurements and comparisons of its query evaluation would make little sense and have not been presented. Arguable, the resulting lack of experimental validation is a shortcoming of the work on XChange^{EQ} so far. However, a physical implementation of query evaluation that is competitive with current data stream management systems would go beyond the scope of this thesis. (Also note that a comparison of XChange^{EQ} and data stream management systems is not straightforward since these systems usually lack certain features such as querying events received as XML messages or reasoning by deductive rules.)

For the future, an implementation of XChange^{EQ} that address the physical level (as well as other related issues such as multi-query optimizations for XML matching, cf. Chapter 19.2.6) is of course highly desirable. As mentioned earlier, both building a query evaluation from scratch and building upon the physical level of existing data stream management systems may be valid options for this.

Chapter 19

Research Perspectives in Complex Event Processing

While the previous chapter has focused on future work that directly relates to XChange^{EQ}, we now broaden the scope to important research directions and perspectives in Complex Event Processing (CEP) in general. Where applicable, we try to point out publications for initial work that has been done on specific topics.

We start with topics that relate to complex event queries (Section 19.1) and their evaluation (Section 19.2) and that are thus closely related to this thesis. Then we discuss topics that go beyond querying of complex events (Section 19.3). Finally we look at topics that relate to the use of CEP in larger contexts (Section 19.4).

19.1 Querying Complex Events

While a number of different query languages for complex events exist (see Chapter 3) and provide solutions to many practical problems (see Chapter 1.1), querying complex events is still a young research topic. Querying complex events in general as well as concrete event query languages open up many new theoretical and practical questions and development directions. In many cases, these questions have counterparts in querying databases; however the fundamental differences between event data that is received over time in streams and non-event data that is readily available in databases or documents require a reconsideration of these questions in the light of complex event queries.

19.1.1 Complexity and Expressiveness

The expressiveness of query languages and the complexity of answering certain queries are topics that are deeply studied in the theory of databases. The results that have been obtained there consider a database query mainly as a mapping from an instance of a database to an answer, i.e., as a simple function for transforming data. This model does therefore not readily accommodate nature of event queries, which in contrast are performed in a step-wise manner over time, taking streams of data as input and producing a stream of data as output.

By using an "omniscient perspective" (like we have done in this work for the declarative and parts of the operational semantics), some results from database queries may transfer in a fairly straightforward manner. However, this omniscient perspective may not be satisfying for studying expressiveness and complexity. The streaming nature of input data gives certain natural restrictions on the expressiveness of event queries, for example, that we cannot query for the absence of events that will be received in the future. Similarly, complexity cannot be studied as the over-all complexity of evaluating an event query, i.e., the sum of the costs of all evaluation steps, because this cost is trivially infinite when the event stream is infinite. Rather we must

study the average or worst case for any individual step or something similar. Such aspects are not covered by the current models of database queries and new models for event queries will be required.

Since event query evaluation involves storing partial histories of events, the space complexity of event queries (given in terms of how much data must be stored across evaluation steps) is a further aspect where we can expect significant differences from traditional database queries. The notion of temporal relevance that his been introduced in Chapter 15 could be considered a first step in this direction. Space complexity for streaming queries is also considered in [ABB+04]; however in this work only regular relational algebra expressions are considered without any notions of temporal windows (as in data stream languages) or temporal conditions (as, e.g., in XChange^{EQ}).

19.1.2 Relevance of Events

Events have to be stored in event query evaluation for as long as they may be relevant for generating future answers to a query. In this work, we have introduced a concrete method for statically determining the temporal relevance of an event, that is, its relevance according to temporal conditions in queries. We have, however, also discussed in Chapter 15.6 that determining the relevance of events is a deeper reaching problem with many variants.

In particular, in many event-driven applications there are natural constraints that restrict how events develop over time and these constraints can be helpful to determine the relevance of events. Because the constraints can be understood as “axioms” (i.e., propositions that we assume to be true) about the event stream, we have termed this “axiomatic relevance” in Chapter 15.6.5.

Investigation of any form of relevance will raise two important questions: How is the information to determine relevance obtained? How is relevance described in a suitable way for query evaluation?

For the temporal relevance of Chapter 15, information for determining relevance has been simply part of queries (in the form of temporal conditions). For axiomatic relevance, the axioms about event streams could be given in different forms: They could be specified explicitly in some special language, ideally one that has a syntax that is close to some event query language (much like integrity constraints in databases). They could also be specified implicitly in descriptions or code for the event sources. Because they provide high-level descriptions, business process specifications would be particularly suitable here. It may also be possible to dynamically try to recognize axioms from event streams as done in [BSW04].

To describe temporal relevance in a way that is suitable for query evaluation, we have employed temporal relevance conditions in Chapter 15. These conditions correspond to simple selection conditions. Axiomatic relevance may require more involved conditions because the relevance of an event there depends not just on attributes of the event but also on the presence or absence of other events. To describe axiomatic relevance in query evaluation it may therefore be necessary to introduce concepts such as (foreign) key joins[1] into the evaluation formalism or extending the notion of relevance conditions beyond selection conditions (e.g., to include nested queries or semi-joins as suggested in Chapter 15.6.4).

19.1.3 Causal and Spatial Event Relationships

In current event query languages, time and order are the only relationships between events that have dedicated constructs for expressing conditions on the events. There is no support for causal or spatial relationships between events, which may be important in many applications.

Causal relationships, which are discussed mainly in [Luc02], allow to reason and express conditions about cause-and-effect chains in event queries. They are especially relevant for event queries that aim at detecting failures and tracking down (potential) causes.

[1]Foreign key join here means a join where we know that each tuple in one of the relations (say the left) will join with at most one other tuple in the other relation (say the right). Accordingly, when a join partner has been found for a tuple on the left, no further processing (i.e., searching for further join partners) is necessary and the tuple need not be stored any longer its event history)

An important aspect for dealing with causality in event queries is how causality information is obtained and represented. It may be represented explicitly in the event data. For example, an event might have an attribute "cause" with an some identifier for another event. Causality information may also be represented separately. For example, there might be a database table to which causality information (e.g., as tuples of event identifiers) is continually added. It may also be the case that causality information must be deduced from event data, business processes, etc. In this last case, a rule language for modeling causality information may convenient. To express, for example that a shipping event is the cause for a delivery event if it happens before and the two events have the same tracking number t, we might write statements such as

$$\text{causes}(s, d) \leftarrow s : \text{shipping}(id, t),\ d : \text{delivery}(t),\ s\ \mathsf{before}\ d$$

Note that while this strawman syntax here is similar to Rel$^{\text{EQ}}$, the difference here is that we have a *predicate* "causes" in the rule head, not an event.

Spatial relationships allow to reason and express conditions involving the positions of events in physical space. They are relevant mainly in applications that processes events from the physical world, where each event is associated with a point or region in space where it occurs. Variations on spatial relationships may however also be relevant in some scenarios that do not relate to the geography of the physical world; for example, when reasoning about events in a distributed network, there are also concepts such as topology or the distance between two network nodes.

There is a wide spectrum for the representation and modeling of space and spatial relationships in different applications. There are relationships based on coordinates (e.g., "located at"), on the topology between regions (e.g., "overlaps" or "includes"), on distance (e.g., "within radius of 10 meters"), on orientation (e.g., "north of"), and much more. Some applications may also require relationships that are derived from spatial characteristics, e.g., the travel time between two event locations (which in turn depends on possible paths and their costs in the physical world). Some first discussions on spatial relationships in event queries can also be found in [Sch08].

While other relationships between events may be conceivable, causal and spatial relationships seem the most relevant and promising in the context of querying events. (Note that we discuss type relationships —such as specialization or generalization through type inheritance— separately in Section 19.3.1, since they are usually not between event instances but between event types or an instance and a type.)

Both causality information and event locations are usually contained in event data. One might therefore argue that they could be queried and reasoned about just like regular data and no dedicated support from the event query languages is required. This may be true, although inconvenient, for simple queries involving very simple relationships (e.g., the causality between shipping and delivery from above or simple spatial relationships that can be expressed with regular arithmetic on coordinates). For more advances queries and relationship, it is not true anymore. Queries involving causality will often involve transitive relationships (e.g., to connect direct causal relations into a causality chain). Queries involving space will require more complicated relationships that also may rely on the topology of the considered space.

Further both causal and spatial relationships may be important for query optimization (much like temporal relationships are, e.g., for garbage collection). Causal relationships typically imply a temporal order between the cause and the effect (due to the "cause-time-axiom" [Luc02]). Spatial relationships can be relevant for a networked query evaluation in a geographically distributed environment, e.g., to minimize communication ways.

With its separation of concerns, XChange$^{\text{EQ}}$ is particularly suited for investigating such relationships. They can be easily added to the language's syntax and semantics in a manner that is consistent with temporal relationships. For example we might then use expressions such as "i causes j" or "$\{i, j\}$ within-radius-of 10 meters" in `where`-clauses (see also Chapter 17.8).

19.1.4 Experimental Studies of Languages

The comparisons of the different event query languages (or language styles) in Chapters 3 and 17 have been conducted in an analytical way, arguing the pros and cons of certain features and

characteristics of the languages. While this is the common way to compare computer languages, it is also unsatisfying: the importance and effect these characteristics have in practice (i.e., in development projects with human programmers) is not clear and not measured in a quantifiable way.

It would therefore be interesting to study and compare different event query languages in an experimental way with real users. Performing such studies in an unbiased way is, of course, not an easy task. The target group for such studies is rather limited (e.g., only persons with programming skills) and has a wide variety in aptitude, skills, and previous knowledge. More generally, computer science does not have as much experience with studies involving human subjects as other disciplines such as psychology. However, the relative novelty of event query languages (compared to regular query languages or programming languages in general) might help for such investigations: at present it is still relatively easy to find users without prior exposure to the topic of querying events and to concrete event query languages.

19.2 Event Query Evaluation and Optimization

Event query languages must provide runtime environments where their queries are evaluated. As for databases and their queries, efficiency of these runtime environments is important and good algorithms and index structures can improve efficency by orders of magnitude. While there has been some amount of research on the efficient evaluation of event queries, especially in the context of data stream systems, still more remains to be investigated.

19.2.1 Benchmarking

An important aspect of research on query optimization are benchmarks, that is, experimental evaluations that measure and compare performance criteria for different query evaluation methods. Note that for event queries there are several important performance criteria such as latency, throughput, memory consumption, and, if approximate query evaluation techniques are used, precision of results. So far, there are no standardized benchmarks for event queries (comparable to say the TPC benchmarks for databases [TPC]) and only little work has been done in this respect so far.

In the context of data stream management systems, the linear road benchmark [ACG+04] has become somewhat established. However it is very specific to the queries in data stream management systems and not so much to the more general scope of CEP. The same is true for the lesser established NEXMark benchmark [NEX]. More recently, a benchmark for CEP engines is developed in the BiCEP project [MBM08, BiC].

In addition to their role for research on query optimization, benchmarks (and especially standardized benchmarks) will also important practically as guidelines that enable developers to choose among different event query engines.

19.2.2 Event Query Planning

Efficient algorithms and data structures to perform the different operations that are involved in query evaluation are an important aspect of query evaluation, and most of the research that has been done so far has been focused on this issue. However, an also very important task in query optimization is to chose the operations that have to be performed (there are typically alternatives) and their order as well as to choose for each operation an algorithm to implement it. This process is usually called query planning.

Query planning, as known from databases, involves exploring alternative query plans (i.e., combinations of operations and their implementations), predicting their performances (also called their costs), and choosing the plan with the best predicted performance. Because the search space of possible query plans can be very large, it is usually explored in a branch-and-bound manner,

where query plans with a good performance are modified in the hope of obtaining a query plan with even better performance.

When trying to transfer the idea of a query planner from databases to event queries, we face two major issues that have already been discussed in Chapter 14.5.5: (1) event queries require different cost measures, often involving trade-offs (e.g., between throughput and latency, or between throughput and precision of results) and (2) these cost measures might rely on statistics and estimations about data distribution, which is not available as readily for event streams as it is for data in a database.

It may be in particular due to the lack of statistics about data distributions that query planning has (to the best of our knowledge) not been explored for event queries so far. On the other hand, there may be ways to obtain statistics by, for example, using histories of event streams to predict their future data distributions. Given the importance of query planning in databases, it certainly should be given more consideration for event queries than it has been so far.

19.2.3 Adaptive Query Evaluation

Adaptive query evaluation (or processing) [DIR07] is closely related to query planning and specifically addresses the lack of statistics about data distributions during query compilation. Adaptive query planning tries to improve the performance of query plans at runtime based on observations made as query evaluation progresses. Example improvements include changing the order of selection operations or (in comparison much harder) changing join order.

Adaptive query evaluation is a fairly recent research topic and so far mainly investigated in a database context. Even though adaptive query evaluation often operates with data streams (as provided, e.g., by an iterator-style query evaluation in a database), little work has been done on applying adaptive query evaluation techniques to complex event queries. Because the evaluation of event queries involves, for example, maintaining event histories together with a garbage collection based on temporal relevance, research will be required to transfer existing adaptive query evaluation techniques from databases to event queries and likely new and different techniques will prove useful.

19.2.4 Specific Main Memory Query Evaluation Techniques

Event query evaluation is usually done in main memory. This is in contrast to classical query evaluation in databases, where data is read from disc.[2] Whereas the primary goal of classical (disc-based) query optimization is to perform few page reads and perform them sequentially, in a main-memory-based query evaluation, sequential access is less relevant and the granularity of data that is read is much smaller (words rather than pages).

There are several optimization techniques for a main-memory-based query evaluation [GS92] that are relevant or appear promising in the context of event queries. These include query planners that take into account CPU time in their cost measures [DKO+84], algorithms that are conscious of processor caches and caching strategies (mainly for performing joins and aggregations) [SKN94, MBK02], and the use of index structures tailored for main memory access [LC86b, LC86a].

19.2.5 Spill-Over from Main Memory to Disc

Although the amount of available main memory is growing, it is still a scarce resource. Especially for event queries that must store some events for a long time (i.e., with high temporal relevance times, cf. Chapter 15), there may still be a need to store some data on disc rather than in main memory. This raises the obvious question which data to store in main memory (if possible) and which data to store on disc (or "spill-over" to disc).

[2] We use the term "classical" here in recognition of the fact that in modern databases query evaluation can also be done in main memory to a large extent and can employ caching to avoid repeatedly reading frequently used data from disc.

Obviously, data that is accessed often in query evaluation should be kept in main memory. Applying this to a relational algebra based evaluation like the one in this thesis, a promising approach may be to partition event histories vertically. Attributes of event tuples that are used in, e.g., in joins are then kept in main memory and attributes that are used, e.g., only in the construction of new results kept on disc. This is particularly interesting in contexts that involve XML event messages, where it may be common that large portions of data from input events are just copied (without further processing) into the query result. An approach where such a technique is applied in the context of streaming evaluation of XQuery (but not an event query context) is discussed in [Sch07].

19.2.6 Efficient XML Matching

In the work on the event query language XChange^{EQ}, we have emphasized that events are increasingly communicated as XML messages and that the data in these XML messages must be queried in event queries. We have called such queries against XML messages simple event queries; recall that complex event queries are built from such simple event queries. In a setting like XChange^{EQ}, we have to conceptually evaluate all simple event queries whenever a new XML message is received to see which of these match the incoming XML message.

In a typical CEP application, the number of such XML queries that are used as simple event queries can be high.[3] When the simple event queries are just evaluated one after another, this can easily become a bottleneck for event query evaluation. This is the case even when efficient methods for evaluating single XML queries are employed.

Because we evaluate a high number of XML queries (in the case of XChange^{EQ}, Xcerpt query terms) against a single XML document, an important way to avoid this bottleneck is to employ multi-query optimization to exploit similarities between different queries. Some approaches have been developed in research for multi-query optimization of XML queries [AF00, DFFT02, Fur03, GGM$^+$04]. These current approaches focus on path queries (as written, e.g., in XPath) and process the incoming XML document in streaming manner (e.g., through a SAX parser) without storing it in main memory.

These assumptions are somewhat different from the assumptions that can be made about XML queries that are used as simple event queries in CEP, however. For typical CEP applications and in particular for uses of the event query language XChange^{EQ}, incoming XML documents can easily be assumed to fit into main memory. Accordingly it is not necessary to process them in a streaming manner and it may even be considered to build index structures for the XML document while parsing it. Further, XML queries against XML messages in CEP applications are usually not just path queries, but more generally tree or even graph queries.

A complement to multi-query optimization of XML queries is their parallelization. Since each simple event query can be evaluated essentially independently from the other queries, matching of simple event queries against incoming XML documents is conceptually fairly easy to parallelize.[4] When multi-query optimizations are involved, parallelization becomes harder because queries are now not independent anymore. An ideal system should therefore be aware of this issue and be able to, e.g., partition queries into independent sets so that as much multi-query optimization as possible is performed in each set but each set is independent of the other sets so that they can be evaluated in parallel.

19.2.7 Multi-Query Optimization for Complex Event Queries

Multi-query optimization is not just relevant for simple event queries. Complex event queries may also have similar subexpressions and exploit multi-query optimization to avoid evaluating these

[3] The use case for Business Activity Monitoring of a very simple order processing application in Chapter 7.1 already uses well over a dozen simple event queries in its rules. In a real application the number can be easily expected to be in the hundreds or even thousands.

[4] Note however that there may be hidden issues such as the fact that the queries might all access the same memory region (that where the XML document is stored) on a multi-processor system.

more than once. As an example, all event queries to detect overdue orders in Chapter 7.1.3 involve an anti-semi-join between order and shipping events when translated into an algebra. It may be beneficial to evaluate this join only once, instead of several times.

The query plans that have been proposed in this work (Chapter 14) enable to describe the sharing of results by introducing additional materialization points. In data stream management systems, a related approach called synopsis sharing is often used [ABB+03]. The more difficult issue however is in the query rewriter and planner that has to recognize equivalent subexpressions in complex event queries. This in turn requires the ability to reason about equivalences of simple event queries (see above).

In general, the equivalence of queries is a problem that has very high complexity or is even undecidable (depending on the expressiveness of the query language). However, multi-query optimization is still possible even if equivalence is undecidable: being able to find only some (but not all) equivalences and using result sharing on them is still better than not performing any result sharing at all.

19.2.8 Distributed Event Query Evaluation

Event queries are often used in the context of distributed computer systems because these naturally use an event-based (or message-based) communication. When events are received from many distributed sources and possibly results of event queries communicated further, a distributed evaluation of event queries may be beneficial. It can minimize communication distances or even the need for communication by pushing query operations closer to the event sources. Additionally, a distributed evaluation will perform operations in parallel and thus may add to performance and scalability of the system.

A reduction of communication and communication distances is not just relevant for saving network bandwidth. Because intelligent sensors in wireless sensor networks are often battery-powered, saving energy is important there. Communication consumes much energy in comparison to other operations (in particular local computations on an embedded processor) and the energy consumption is proportional to the distance.

19.2.9 Garbage Collection

An important aspect in event query evaluation that is not found in database or other non-event queries is garbage collection of events that become irrelevant. We have already discussed this issue in Section 19.1.2 from the perspective where we determine relevance conditions that describe whether an event is still relevant or not at runtime.

At runtime, relevance conditions such as the temporal relevance conditions that have been introduced in Chapter 15 must be evaluated as part of the query evaluation. To make their evaluation efficient appropriate index structures must be investigated. Note that it is not just sufficient to maintain an index to locate irrelevant events quickly. Garbage collecting an irrelevant event also entails updating all other indexes and storage structures that include the irrelevant event (e.g., if there is a hash join involving the irrelevant event then the event must also be removed from the corresponding hash index by the garbage collector).

A related question is when to perform garbage collection. In general, an event may still be stored without effect on the query results even if it has become irrelevant. Therefore it is not necessary to immediately remove an event when it has become irrelevant. Garbage collection therefore need not be performed as part of each evaluation step (as in the conceptual query evaluation algorithm of Chapter 15.5). It may be performed in an independent thread at different times. This allows, for example, to perform garbage collection at regular intervals, to perform it on demand when memory becomes scarce, or to perform it at convenient times when the processor load from regular query evaluation is low.

19.3 Beyond Querying of Complex Events

We know leave the realm of event queries and their evaluation and turn to topics where complex events are processed with means that go beyond querying.

19.3.1 Knowledge Representation for Events

Current event query languages process events as plain data without much knowledge —or meta-data— about them. Meta-data and knowledge representation for events may contribute to a more intelligent processing of events, much in the fashion that meta-data and knowledge representation promise to do for regular, non-event data.

In addition to what deductive rules about events (like those in XChange$^{\mathrm{EQ}}$) may provide, other forms of knowledge representation may be relevant. It may be useful to have (event) type hierarchies to express specializations, generalizations, and other relationships between event types. For example, one might specify that both fire alarms and flood alarms are alarms. With this knowledge, reactive or deductive rules about alarms may automatically also apply to the more special fire alarms or flood alarms. More advanced formalisms such as ontologies may be useful to represent more advanced knowledge about events and their relationships. For example, one might express a statement that each alarm must have at least one cause, which in turn must be a sensor event.

Knowledge representation for events may also go beyond relationships between event types. For example one might express that the person that is notified upon an alarm must be knowledgeable about first aid. A statement like this involves not only concepts relating to events and their types (here: alarm) but also other, non-event concepts such as person or first aid.

The general promise of meta-data and knowledge representation for events is the same as for regular data: one hopes to create more intelligent systems that are easier to develop and maintain by using appropriate formalisms to represent knowledge explicitly and in a modifiable way rather than hiding it in code. (Event) data integration, which may be necessary due to heterogeneity in event data formats and representation, may become easier when appropriate knowledge representation formalisms are used. Knowledge about events and their relationships with other concepts in an event-based system may also be important for validating or verifying the behavior of the system.

19.3.2 Uncertainty and Probability in Querying Events

Event querying with current languages is done in a precise and crisp fashion: Data in events has precise values. A simple or complex event either happens or it does not. Temporal conditions on events are either true or false.

However, when event queries process data from the physical world, e.g., from sensors, the simple events that provide input data do not satisfy this precision. There may be errors in measurements and also doubts in whether an event happened at all. The imprecision in the input may carry over to the output, that is, to complex events.

By combining the values of many sensor, CEP can help reducing the imprecision in output in a process often termed sensor data cleaning. However, since current event query languages are built to work with precise and crisp events and data, they are not as suitable for this task as they might be. For example, one can try to minimize the detection of false fire alarms by detecting outliers in or averaging over sensor data. One can however not represent uncertainties in the complex events that are deduced. For example, one cannot express directly that there is a 90% chance of a fire outbreak.

Representing probabilities and uncertainties is especially important when applying further reasoning to deduced events. For example from a 90% chance of a fire outbreak and a 10% chance of the sprinklers not having been activated, we could deduce that there is a 9% chance that we have an uncontained fire. Probabilities and uncertainties are also important in systems that aim

at predicting the further development of events (much like a weather forecast does) and possibly take countermeasures.

There have been some recent approaches on dealing with uncertainty and probabilities in querying events, for example in [WGET08] and [RLBS08]. So far, these works primarily investigate issues related to query evaluation and complexity; design of a language can be said to not having been a concern so far. Rule-based languages for event processing with model-theoretic semantics such as XChange^{EQ} seem particularly well-suited as candidates for processing uncertain and probabilistic events: there is significant experience with different forms of uncertainty for traditional non-event rule languages that can be built upon.

19.3.3 Event Mining

Event query languages address the issue of detecting combinations of event occurrences where the combination is known and described a priori in the query. This works well in applications such as Business Activity Monitoring or processing of sensor data where we have a clear conception of what patterns of events we are looking for. It does not however work in applications where we do not have such a clear conception and cannot specify a query (or pattern) a priori.

Such an application where we want to detect patterns of event occurrences that are *unknown*, i.e., not given as a query, is fraud detection for credit cards [WvASW07]. More generally, unknown patterns are interesting in any application involving the detection of irregular behavior through monitoring of events, be it intrusion detection in networks, supervision of financial transactions, or health-related monitoring of body functions such as temperature or heart rate.

The discovery of unknown information or knowledge has been given much consideration for regular, non-event data with disciplines like data mining and machine learning. However, there has been little work in applying and evaluating these approaches for event streams, where data is received over time in a step-wise manner and where a timely detection matters. In allusion to data mining, we call this discovery of unknown information or knowledge here "event mining."

The problem of detecting irregular behavior in event streams could be understood as a problem of outlier detection as known from data mining. However, it is somewhat different because outliers in outlier detection are defined as single data points; in event processing, we however care mainly about irregular combinations of events not single irregular events.

There may also be further opportunities for event mining beyond the application of data mining and machine learning techniques to detect irregular behavior in event streams. For example, clustering techniques may be useful to detect events that commonly occur jointly (i.e., within short time of each other). This is turn may help in optimizing business processes that generate these events or redistributing sensors in a physical environment.

19.4 Complex Event Processing an a Larger Context

In this final section, we turn to research perspectives that are not directly within the realm of CEP, i.e., that are not concerned with processing of events, but that address the use of CEP in a larger context.

19.4.1 Detection and Generation of Simple Events

The fairly obvious requirement to preform CEP is to have the ability to detect or receive simple events. Simple events, however, do not appear from nowhere; they must be generated by the event sources and communicated to the CEP engine. Not only requires this a cooperation of the event sources, it also entails that the programmers of components that may become event sources have a clear idea of which events may be relevant to other components in the systems (the CEP engine in particular).

The first issue this raises is how to develop software systems so that the events that might be relevant are generated. Further, we are typically are not be able to anticipate all events that

might become relevant during a system's lifetime in the initial development phase. It is therefore also important to architect such systems so that we can easily modify code to add the generation of new events.

Generation of events in traditional imperative and object-oriented programming languages can lead to fairly tangled code: the event generation is not part of the basic functionality of the system but must be mixed into code for basic functionality. For example, in an online shop application the basic functionality of the system is to move data around between Web forms and different databases to take and process orders. Generation of events that may be useful, e.g., for Business Activity Monitoring, will be mixed into this code although it does not directly contribute to taking and processing orders.

There are currently at least two promising approaches to tackle this issue. Higher-level languages tailored for the description of workflows and business processes like BPEL [Hav05] may automatically generate events, e.g., at state transitions. Aspect-Oriented Programming (AOP) [KLM+97] seeks to separate code concerning basic functionality from cross-cutting code that concerns other aspects such as security, logging, or said generation of events.

The second issue that the need for simple events raises is how to add the generation of events to legacy systems that have not been architected with event generation in mind and where we might not have access to or be able to modify the source code. Solutions to this issue often involve a fair amount of low-level wizardry such as monitoring system output in log files or temporary files (often using operating system functions to detect file modifications) or using database triggers to detect and intercept data modifications that indicate relevant events.

19.4.2 Push and Pull Communication of Events

Current CEP systems assume that events are communicated in a push manner. That is, the system is automatically informed whenever a relevant event occurs. While this works well in many scenarios, there are also scenarios where events must be retrieved in a pull manner. In those scenarios, a push communication of events may not be possible or have disadvantages, e.g., relating to network connectivity or bandwidth.

An example where push communication usually is not possible is monitoring data sources on the Web such as HTML documents or RSS feeds. Updates to these data sources can be understood as events and there may be a need for querying complex events there. However, typical Web data sources can only be retrieved in a pull manner and do not inform users when they are changed. To process events from these sources, therefore, users must periodically retrieve them. Retrieval of e-mails with protocols such as IMAP [Cri03] or POP3 [MR96] is another example where events (in this case e-mails) can only be retrieved in a pull manner.

Periodically retrieving some data source to detect events immediately leads to the question how often the data source should be retrieved. Typically this is a trade-off between a need for timeliness of information and the usage of resources such as network bandwidth. The need for timeliness of information in turn is influenced by the processing that is applied to events, concretely for example complex event queries (and possible temporal conditions in them), and the rate in which updates are performed to the data source. Note that the update rate is in general unknown but one might attempt to guess it by looking at historical information. Approaches that adaptively set the time intervals between periodic retrievals of data sources are investigated in [BGR06] and [RGR08].

19.4.3 CEP Design Patterns

Since CEP is still a new technology, developers often lack the experience to successfully engineer systems that are based on or include CEP. Clearly there is therefore a need to document guidelines to educate developers about CEP and its uses in software systems. Design patterns [GHJV95] have been proven successful for documenting common problems and good solutions. Design patterns are primarily rooted in object-oriented design and programming, but the pattern-based approach has since then also entered other disciplines such as software architecture in general [BMR+96], enterprise application architecture [Fow02], or user interface design.

While there has been some work on design patterns in event-driven architectures [HW03], little work has been done so far on design patterns that relate explicitly to CEP. Given that CEP is still in its early stages, this in unsurprising since there are still not so many successful CEP projects to elicit design pattern from. However there are recent efforts to start collecting and categorizing CEP-related design patterns [Pas08, PvA08] as well as more focused design patterns for event queries [Cor07].

19.4.4 Development Tools and Visualization

The success of CEP as a technology will not only depend on the availability of expressive and easy-to-use event query languages, other complex event detection facilities, and experience in software architecture. Like other technologies, success of CEP will require an ecosystem of tools for development. This may include support for CEP languages and components in integrated development environments (IDEs), editors for event queries that support features such as syntax highlighting or auto completion, graphical editors for event queries, visualization of event types and type hierarchies, tools for measuring performance, debuggers for event queries and other CEP-related components, and tools for validating and verifying event queries and CEP systems with respect to higher-level specifications.

Current prototypes and products in the CEP area focus on event query languages and their runtime engines, providing little tool support so far as discussed in Chapter 3.5.3. In [BMH99], visualizations of the detection process of complex events are discussed in the context of a composition-operator-based language. Such visualizations may be useful for example to understand and debug complex event queries. Formal verification of complex event queries by means of model checking is discussed in [EB06, EPBS07], again in the context of a composition-operator-based language.

Chapter 20

Summary and Conclusion

In this work we have investigated practical and theoretical issues related to querying complex events. Guided by the development of the high-level event query language XChange^{EQ}, we have covered the spectrum from language design over declarative semantics to operational semantics.

At the heart of the language design of XChange^{EQ} is the idea that the four querying dimensions data extraction, event composition, temporal and other relationships between events, and event accumulation must be separated. This, together with its support for deductive rules and for querying events represented as XML messages, arguably makes XChange^{EQ} more expressive and easier to use than other event query languages.

With its model theory and accompanying fixpoint theory, XChange^{EQ} provides highly declarative semantics for querying events based on an approach that is well-established for traditional, non-event queries. Importantly, these semantics work directly on streams and do not require a concept of state, which makes them also very intuitive. A core idea in applying this traditional approach from non-event query languages to an event query language is to pretend "omniscience," that is, we first assume that we know the full, infinite event stream and specify semantics irrespective of the evaluation times of event queries. Only then we show that these semantics are actually designed to work for a streaming evaluation over time, where we only now the history of events received so far.

Operational semantics of XChange^{EQ} are based on CERA, a variant of relational algebra that meets both the expressiveness requirements and the restrictions that are needed for evaluating event queries incrementally, and on the notion of query plans with materialization points. Again, a core idea in the translation of XChange^{EQ} programs into algebra expressions and query plans is to pretend "omniscience" at first, where we ignore the step-wise nature of evaluation over time. Only then we attend to the incremental evaluation of query plans by applying finite differencing.

Since evaluation of event queries involves storing and maintaining histories of events received so far as well as some intermediate results, there is a need for garbage collection. To enable garbage collection in the evaluation of XChange^{EQ}we have defined the notion of relevance and developed an algorithm for determining temporal relevance based on temporal conditions in queries. Temporal relevance also may play an important role for query planning.

Complex Event Processing is an emerging and exciting research area and event query languages play an important part in this area. This work on XChange^{EQ} has contributed to this field by going new ways in its language design and investigating formal foundations that have strong roots in the tradition of databases and logic programming. There are, however, still many unsolved issues in querying and, more generally, processing complex events.

Part VI

Appendix

Appendix A

EBNF Grammars

The following provides context-free grammars for XChange^{EQ} and its "little brother" Rel^{EQ} in EBNF notiation

A.1 Conventions on EBNF Notation

The EBNF notation used here follows the conventions of the XML 1.1 Recommendation [B+06b, Section 6]:

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

A.2 (Core) Xcerpt Term Grammar

The following is a simplified grammar for data, query and construct terms of Xcerpt. It skips details such as XML namespaces, graph-like references, identity variables, position specification, optional subterms, order for groupings, regular expression matching, etc. that are not relevant for the understanding of XChange^{EQ}.

[1] This is a slight deviation from the notation of [B+06b] used in order to enhance readability.

```
DataTerm      ::= DTLabel DTChildren
                | '"' String '"'
DTChildren    ::= "{"  ( DataTerm ","? )* "}"
                | "["  ( DataTerm ","? )* "]"
DTLabel       ::= String

QueryTerm     ::= Label QTChildren
                | '"' String '"'
                | Variable
                | Variable "->" QueryTerm
                | "desc" QueryTerm
                | "without" QueryTerm

Label         ::= Variable
                | String
QTChildren    ::= "{"  ( QueryTerm ","? )* "}"
                | "{{" ( QueryTerm ","? )* "}}"
                | "["  ( QueryTerm ","? )* "]"
                | "[[" ( QueryTerm ","? )* "]]"
Variable      ::= "var" Identifier

ConstructTerm ::= Label CTChildren
                | '"' String '"'
                | Variable
                | Grouping
                | Aggregation "(" Grouping ")"
                | Function "(" CounstructTerm ("," ConstructTerm)* ")"

CTChildren    ::= "{"  ( ConstructTerm ","? )* "}"
                | "["  ( ConstructTerm ","? )* "]"
Grouping      ::= "all" ConstructTerm ("group by" "{" Variable+ "}")?
Aggregation   ::= "min" | "max" | "sum" | "count" | "avg"
Function      ::= "div" | "mult" | "add" | "subtract" | "mod" | "concat"
```

A.3 XChange^{EQ}Grammar

We now give the grammar for XChange^{EQ}. The start symbol is `Program`. Note that the grammar makes use of `QueryTerm`s and `ConstructTerm`s as defined in the previous section.

```
Program       ::= (DeductiveRule | ReactiveRule)*
DeductiveRule ::= "DETECT" ConstructTerm "ON" BodyEQ "END"
ReactiveRule  ::= "RAISE" RaiseSpec "ON" BodyEQ "END"
RaiseSpec     ::= "to" "(" "recipient" "=" '"' String '"' ","
                           "transport" "=" '"' String '"' ")"
                    "{" ConstructTerm "}"

BodyEQ        ::= SimpleEQ
                | "and" "{" (EQ ","?)+ "}"
                | "or"  "{" (EQ ","?)+ "}"
                | EQ "where" "{" (Condition ","?)+ "}"
EQ            ::= BodyEQ
                | WhileEQ
                | RelTimerEQ
SimpleEQ      ::= "event" Identifier ":" QueryTerm
```

```
WhileEQ       ::= "while" Identifier ":" ("not"|"collect") QueryTerm
RelTimerEQ    ::= RelTimerSpec "[" "event" Identifier, Duration "]"
Condition     ::= DataCondition
                | TempCondition

RelTimerSpec  ::= "extend" | "shorten"
                | "extend-begin" | "shorten-begin"
                | "shift-forward" | "shift-backward"

DataCondition ::= Expr CompOp Expr
CompOp        ::= "<" | "<=" | "=" | ">=" | ">" | "!="
Expr          ::= Variable
                | '"'String'"'
                | Number
                | Expr ArithOp Expr
                | "(" Expr ")"
ArithOp       ::= "*" | "/" | "+" | "-" | "mod"

TempCondition ::= Identifier AllenOp Identifier
                | "{" (Identifier ","?)+ "}" "within" Duration
                | "{" Identifier "," Identifier "}" "apart-by" Duration
AllenOp       ::= "before"     | "contains"    | "overlaps"
                | "after"      | "during"      | "overlapped-by"
                | "starts"     | "finishes"    | "meets"
                | "started-by" | "finished-by" | "met-by"
                | "equals"
Duration      ::= ( Number ("week"|"weeks") )?
                  ( Number ("day"|"days") )?
                  ( Number ("hour"|"hours") )?
                  ( Number "min" )?
                  ( Number "sec" )?
                  ( Number "ms" )?          /* constraint: not empty! */
```

A.4 Rel^{EQ} Grammar

The following is a grammar for Rel^{EQ}. The start symbol is again `Program`.

```
Program     ::= Rule*
Rule        ::= Head "<-" EQ ";"

HeadAtom    ::= PredName "(" ( HeadTerm ("," HeadTerm)* )?  ")"
HeadTerm    ::= Variable | Grouping | '"' String '"' | Number
Variable    ::= Identifier
Grouping ::= ( "min" | "max" | "sum" | "count" | "avg" ) "(" Variable ")"

EQ          ::= SimpleEQ ("," SimpleEQ)*
                ("," RelTimerEQ)* ("," WhileEQ)* ("," Condition)*
SimpleEQ    ::= Identifier ":" BodyAtom
RelTimerEQ  ::= RelTimerSpec "(" Identifier, Duration ")"
WhileEQ     ::= "while" Identifier ":" ("not"|"collect") BodyAtom
BodyAtom    ::= PredName "(" ( BodyTerm ("," BodyTerm)* )? ")"
BodyTerm    ::= Variable | '"' String '"' | Number
Condition   ::= DataCondition
              | TempCondition
```

```
RelTimerSpec  ::= "extend" | "shorten"
                | "extend-begin" | "shorten-begin"
                | "shift-forward" | "shift-backward"

DataCondition ::= Expr CompOp Expr
CompOp        ::= "<" | "<=" | "=" | ">=" | ">"
Expr          ::= Variable | Number

TempCondition ::= Identifier AllenOp Identifier
                | "{" (Identifier ","?)+ "}" "within" Duration
                | "{" Identifier "," Identifier "}" "apart-by" Duration
AllenOp       ::= "before"     | "contains"    | "overlaps"
                | "after"      | "during"      | "overlapped-by"
                | "starts"     | "finishes"    | "meets"
                | "started-by" | "finished-by" | "met-by"
                | "equals"
Duration      ::= ( Number "w" )? ( Number "d" )? ( Number "h" )?
                  ( Number "m" )? ( Number "s" )? ( Number "ms" )?
                  /* constraint: not empty! */
                | Number
```

Appendix B

Proofs about Operational Semantics

B.1 Temporal Preservation of CERA

The proof of the temporal preservation property of CERA of Chapter 13.3 is a simple structural induction on the expression E with a case for each operator. By design of CERA, each operator is restricted in such a way that it maintains the temporal preservation property; this makes the proof fairly simple so that we sketch the ideas behind each case only shortly here.

The base case, where $E = R_1$ is just a single relation, is trivial: just expand $\sigma[M_E \leq \mathit{now}](E)$ to its definition.

For selection $E = \sigma_C(F)$, renaming $E = \rho_R(F)$, natural join $E = F \bowtie G$, and projection $E = \pi_A(F)$ (with preservation of time stamp attributes) the claim is also obvious: they do not change the time stamps of their input at all. Therefore the occurrence times are not affected. Note that in the case of projection this only because we have restricted it in a way so that time stamp attributes must not be dropped.

Similarly, in the case of grouping $E = \gamma_{A,a \leftarrow F(A)}(F)$, the restriction that all time stamp attributes must be grouping attributes ensures that all time stamps are preserved. Note that it is not important what the aggregation function $F(A)$ in the grouping is.

In the case of matching $E = \mathsf{Q}^{\mathsf{X}}_{i:q}(F)$, the time stamps of the result are "inherited" from the input and thus also preserved.

In the case of merging $E = \mu_{i \leftarrow j_1 \sqcup \cdots \sqcup j_n}(F)$, time stamps are changed. However, the theorem only cares about the occurrence time which is the maximum over the time stamps. This is, by definition of the operator, not changed since the new time stamp that is generates is the maximum over the input time stamps.

Similarly, in the case of the temporal anti-semi-join $E = F \overline{\ltimes}_{i \sqsupseteq j} G$, the temporal condition $i \sqsupseteq j$ ensures that the occurrence time of the right input tuples is lower than the occurrence time of the left input tuples; thus result tuples have the occurrence time of the left input tuples.

Note that construction C^{X} is not an operator but an aggregation function; therefore it need not be considered in the induction.

B.2 Correspondence between Relations and Σ, τ

The following completes the proof of the correctness lemma of Chapter 13.4.3.

We first define $\Sigma_{S'}, \tau_{S'}$ and $S'_{\Sigma,\tau}$. The function ref is the function used for generating event references in the definition of the matching operator Q^{X}. Since ref is injective, we can use its inverse, which we denote ref^{-1}. Recall in the following definitions that F is the subexpression of the rule body that is being translated by the CERA expression S.

$\Sigma_{S'} := \{\sigma \mid \exists s' \in S' \forall X \in sch_{data}(S')\ \sigma(X) = s'(X)\}$

$$\tau_{S'}(i) = \begin{cases} ref^{-1}(s'(i.ref))^{[s'(i.s),s'(i.e)]} & \text{if } i \text{ event identifier of a simple event query in } F, \\ \tau_{S'}(j)^{[s'(i.s),s'(i.e)]} & \text{if } i \text{ event identifier of a relative timer event in } F \\ & \quad \text{defined relative to } j, \\ \bot & \text{otherwise} \end{cases}$$

where s' some tuple in S'(choice does not matter)

$$\begin{aligned} S'_{\Sigma,\tau} = \{s' \mid \exists \sigma \in \Sigma.\ & \forall X \in sch_{data}\ s'(X) = \sigma(X), \\ & \forall i.ref \in sch_{ref}\ s'(i.ref) = ref(\tau(i)), \\ & \forall i.s \in sch_{start}\ s'(i.s) = begin(\tau(i)), \\ & \forall i.e \in sch_{start}\ s'(i.e) = end(\tau(i))\} \end{aligned}$$

The lemma is shown for $B_1, \dots, B_n$ by induction on n. Then it is additionally also shown for C. We sketch the individual cases shortly, making the first one a bit more detailed and concentrating more on the underlying idea with the others.

Simple Event Query B_1

Soundness: S' has been produced by the matching operator in B_1. Accordingly, there must be an $e^t \in E$ so that $\Sigma_{S'}(q) \preceq e$ because of the way the matching function is defined. Further, $\tau_{S'}(j_1) = e^t$ because of the way the matching operator generates event references and time stamps. With that $E, \Sigma_{S'}, \tau_{S'} \models b_1$.

Completeness: From $E, \Sigma, \tau \models F^t$ it follows that there is an $e^t \in E$ such that $\Sigma(q) \preceq e$ and $\tau(j_1) = e^t$ by the definition of the model theory. Applying the definition of the matching operator $\mathsf{Q}^{\mathsf{X}}_{j_1:q}$ to e^t shows us that $S'_{\Sigma,\tau} \subseteq S$ and $t = occtime(S')$.

Further Simple Event Queries (B_i, $1 < i \leq k$)

The same reasoning as earlier applies to b_i. By induction hypothesis, completeness and soundness hold for B_{i-1}. It remains to convince ourselves that the natural join in the algebra corresponds to the conjunction $\wedge$ of the model theory (and in particular that it maintains that Σ is maximal for the soundness). Because the joins combines exactly those tuples that agree on the shared attributes (which in turn correspond to shared variables) and because so far there are no negated variables, this should be clear.

Relative Timer Events (B_i, $k < i \leq l$)

Soundness: The bundles in B_i are the same as in B_{i-1}, only tuples in each bundle are augmented with the additional time stamps j_i. Therefore $\Sigma_{S'}$ is the same as the one for the corresponding bundle of S' in B_{i-1} (and, of course, remains maximal). The rest is just the way $\tau_{S'}$ has been defined above.

Completeness: Similarly, we only have to look at τ and convince ourselves that the time stamps in $S'_{\Sigma,\tau}$ are correct. They are because of the correspondence between the auxiliary relations and the definitions of the model theory.

Event Accumulation for Collection (B_i, $l < i \leq m$)

This time, τ remains unaffected. This is why it was important that we drop $i'.ref$ with the projection in B_i. The temporal condition of the join ensures that additional attributes are joined with the right tuples as prescribed by the `while` part of b_i.

Event Accumulation for Negation (B_i, $m < i \leq n$)

Again τ remains unaffected. Note that the anti-semi-join already drops the event reference $i'.ref$, so that no extra projection was required as in the previous case. The anti-semi-join removes those tuples from each bundle in B_{i-1} that would satisfy the negated query of b_i. The temporal condition ensures that this is done only with the temporal restriction of the `while` part of b_i.

Conditions (C)

The bundles in C remain unaffected compared to B_n and τ remains unaffected. The selection simply removes all the tuples that would not satisfy the conditions of the where clause, which is exactly the same as required by the model theory.

Bibliography

[AA07] José Júlio Alferes and Ricardo Amador. r3: A foundational ontology for reactive rules. In *Proc. Int. Conf. on Ontologies, DataBases, and Applications of Semantics*, volume 4803 of *LNCS*, pages 933–952. Springer, 2007.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29:162–194, 2004.

[ABB+07] Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Jendrik Johannes. Modular Web queries — from rules to stores. In *Proc. Int. Workshop on Scalable Semantic Web Knowledge Base Systems*, volume 4806 of *LNCS*, pages 1165–1175. Springer, 2007.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[AC05] Raman Adaikkalavan and Sharma Chakravarthy. Formalization and detection of events using interval-based semantics. In *Proc. Int. Conf. on Management of Data (COMAD)*, pages 58–69. Computer Society of India, 2005.

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 1(59):139–165, 2006.

[ACÇ+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proc. Int. Conf. on Very Large Databases*, pages 480–491. Morgan Kaufmann, 2004.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[AE04] Asaf Adi and Opher Etzion. Amit — the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. Int. Conf. on Very Large Databases*, pages 53–64. Morgan Kaufmann, 2000.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[ANTa] ANTLR IDE — An eclipse plugin for ANTLRv3 grammars. `http://antlrv3ide.sourceforge.net/`.

[ANTb] ANTLR Parser Generator. `http://www.antlr.org`.

[ANTc] ANTLR v3 task for Ant. `http://antlr.org/share/1169924912745/antlr3-task.zip`; linked also from `http://antlr.org/share/list`.

[Apa] Apache Software Foundation. Apache Ant. `http://ant.apache.org/`.

[ASSC02] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, 2002.

[B+06a] Tim Bray et al. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, World Wide Web Consortium, 2006.

[B+06b] Tim Bray et al. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, World Wide Web Consortium, 2006.

[B+07a] Anders Berglund et al. XML path language (XPath) 2.0. W3C recommendation, World Wide Web Consortium, 2007.

[B+07b] Scott Boag et al. XQuery 1.0: An XML query language. W3C recommendation, World Wide Web Consortium, 2007.

[BBB+07] Bruno Berstel, Philippe Bonnard, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactive rules on the Web. In *Reasoning Web, Int. Summer School*, volume 4636 of *LNCS*, pages 183–239. Springer, 2007.

[BBEP05] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 187–192. Springer, 2005.

[BBFS05] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web query languages: A survey. In *Reasoning Web, Int. Summer School*, volume 3564 of *LNCS*, pages 35–133. Springer, 2005.

[BBS03] Sacha Berger, François Bry, and Sebastian Schaffert. A visual language for Web querying and reasoning. In *Proc. Int. Workshop on Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, pages 99–112. Springer, 2003.

[BBSW03] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *Proc. Int. Conf. on Very Large Databases (Demonstations)*, pages 1053–1056. Morgan Kaufmann, 2003.

[BC06] Roger S. Barga and Hillary Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 919–930. Springer, 2006.

[BE05a] François Bry and Michael Eckert. Processing link structures and linkbases in the Web's open world linking. In *Proc. ACM Conf. on Hypertext and Hypermedia*, pages 135–144. ACM, 2005.

[BE05b] François Bry and Michael Eckert. Processing link structures and linkbases on the Web. In *Proc. Int. Conf. on World Wide Web, posters*, pages 1030–1031. ACM, 2005.

[BE06a] François Bry and Michael Eckert. A high-level query language for events. In *Proc. Int. Workshop on Event-driven Architecture, Processing and Systems*, pages 31–38. IEEE, 2006.

[BE06b] François Bry and Michael Eckert. Twelve theses on reactive rules for the Web (invited paper). In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 842–854. Springer, 2006.

[BE07a] François Bry and Michael Eckert. Rule-Based Composite Event Queries: The Language XChange^{EQ} and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *LNCS*, pages 16–30. Springer, 2007.

[BE07b] François Bry and Michael Eckert. Temporal order optimizations of incremental joins for composite event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*. ACM, 2007.

[BE07c] François Bry and Michael Eckert. Towards formal foundations of event queries and rules. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.

[BE07d] François Bry and Michael Eckert. Twelve theses on reactive rules for the Web. In *Proc. Dagstuhl Seminar Event Processing*, number 07191 in Dagstuhl Seminar Proceedings. IBFI, 2007.

[BE08a] François Bry and Michael Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 289–300. ACM, 2008.

[BE08b] François Bry and Michael Eckert. Rules for making sense of events: Design issues for high-level event query and reasoning languages (position paper). In *Proc. AAAI Spring Symposium AI Meets Business Rules and Process Management*, number SS-08-01 in AAAI Technical Reports, pages 12–16. AAAI Press, 2008.

[Bec04] Dave Beckett. RDF/XML syntax specification (revised). W3C recommendation, World Wide Web Consortium, 2004.

[BEE^{+}07] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of rule-based query answering. In *Reasoning Web, Third International Summer School 2007*, volume 4636 of *LNCS*, pages 1–153. Springer, 2007.

[BEFL08] François Bry, Norbert Eisinger, Tim Furche, and Benedikt Linse. Simulation subsumption or déjà vu on the Web. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, LNCS. Springer, 2008. To appear.

[BEGP06a] François Bry, Michael Eckert, Hendrik Grallert, and Paula-Lavinia Pătrânjan. Evolution of distributed Web data: An application of the reactive language XChange. In *Proc. Int. Conf. on Data Engineering (Demonstrations)*, 2006.

[BEGP06b] François Bry, Michael Eckert, Hendrik Grallert, and Paula-Lavinia Pătrânjan. Reactive Web rules: A demonstration of XChange. In *Proc. Int. Conf. on Rules and Rule Markup Languages (RuleML) for the Semantic Web, Posters and Demonstrations*, 2006.

[BEP06a] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*, volume 3842 of *LNCS*, pages 38–47. Springer, 2006.

[BEP06b] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1):3–24, 2006.

[BEP06c] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. XChange: Rule-based reactivity for the Web. In Miltiadis Lytras, editor, *Semantic Web Fact Book*. AIS SIGSEMIS, 2006.

[BEPR06] François Bry, Michael Eckert, Paula-Lavinia Pătrânjan, and Inna Romanenko. Realizing business processes with ECA rules: Benefits, challenges, limits. In *Proc. Int. Workshop on Principles and Practice of Semantic Web*, volume 4187 of *LNCS*, pages 48–62. Springer, 2006.

[Ber02] Bruno Berstel. Extending the RETE algorithm for event management. In *Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 49–51. IEEE, 2002.

[BFB+05] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web reconsidered: Design principles for versatile Web query languages. *Int. J. on Semantic Web and Information Systems*, 1(2):1–21, 2005.

[BFF+07] Irina Botan, Peter M. Fischer, Daniela Florescu, Donald Kossmann, Tim Kraska, and Rokas Tamosevicius. Extending XQuery with window functions. In *Proc. Int. Conf. on Very Large Data Bases*, pages 75–86. ACM, 2007.

[BFHL07] François Bry, Tim Furche, Alina Hang, and Benedikt Linse. GRDDLing with Xcerpt: Learn one, get one free! In *Proc. European Semantic Web Conf., Demonstrations*, 2007.

[BFLP08] François Bry, Tim Furche, Benedikt Linse, and Alexander Pohl. Xcerpt$^{\text{RDF}}$: A pattern-based answer to the semantic web challenge. In *Proc. Int. Workshop on (Constraint) Logic Programming*, 2008. To appear.

[BFLS06] François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Efficient evaluation of n-ary conjunctive queries over trees and graphs. In *ACM Int. Workshop on Web Information and Data Management*, pages 11–18. ACM, 2006.

[BFMS06] Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert. An ECA engine for deploying heterogeneous component languages in the Semantic Web. In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 887–898, 2006.

[BG04] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, World Wide Web Consortium, 2004.

[BGR06] Laura Bright, Avigdor Gal, and Louiqa Raschid. Adaptive pull-based policies for wide area data delivery. *ACM Transactions on Database Systems*, 31(2):631–671, 2006.

[BiC] BiCEP. `http://bicep.dei.uc.pt/`.

[BK08a] Harold Boley and Michael Kifer. RIF basic logic dialect. W3C working draft, World Wide Web Consortium, 2008.

[BK08b] Harold Boley and Michael Kifer. RIF framework for logic dialects. W3C working draft, World Wide Web Consortium, 2008.

[BKK04] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite events for XML. In *Proc. Int. Conf. on World Wide Web*, pages 175–183. ACM, 2004.

[BKPP07] Harold Boley, Michael Kifer, Paula-Lavinia Patranjan, and Axel Polleres. Rule interchange on the Web. In *Reasoning Web, Int. Summer School*, volume 4636 of *LNCS*, pages 269–309. Springer, 2007.

[BL99] Tim Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. Harper, 1999.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, The Internet Society, 2005.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[BLMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (URL). RFC 1738, The Internet Society, 1994.

[BLO+08] François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, Martin Roeder, and Marc Weinberger. The Facility Control Markup Language FCML. In *Proc. Int. Conf. on the Digital Society*, pages 117–122, 2008.

[BMH99] Mikael Berndtsson, Jonas Mellin, and Urban Högberg. Visualization of the composite event detection process. In *Proc. Workshop on User Interfaces to Data Intensive Systems*, pages 118–127. IEEE Computer Society, 1999.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns 1.* Wiley & Sons, 1996.

[Bol05] Oliver Bolzer. Towards data-integration on the semantic web: Querying RDF with Xcerpt. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2005.

[BRS05] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar types and constraints for Web applications. In *Proc. Int. World Wide Web Conf.*, pages 702–711. ACM, 2005.

[Bry90] François Bry. Query evaluation in deductive databases: Bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[BS03] François Bry and Sebastian Schaffert. An entailment relation for reasoning on the Web. In *Proc. Int. Conf. on Rules and Rule Markup Languages (RuleML)*, volume 2876 of *LNCS*, pages 17–34. Springer, 2003.

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29(3):545–580, 2004.

[BW03] David Bailey and Edwin Wright. *Practical SCADA for Industry*. Newnes, 2003.

[BZBW95] Alejandro P. Buchmann, Jürgen Zimmermann, José A. Blakeley, and David L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proc. Int. Conf. on Data Engineering*, pages 117–128. IEEE, 1995.

[CA08] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: Harnessing and unleashing their synergy! In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 1–12. ACM, 2008.

[CCC07] K. Mani Chandy, Michel Charpentier, and Agostino Capponi. Towards a theory of events. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 180–187. ACM, 2007.

[CD99] James Clark and Steve DeRose. XML path language (XPath) version 1.0. W3C recommendation, World Wide Web Consortium, 1999.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2001.

[CER] CERN Colt Scientific Library 1.2.0. `http://dsd.lbl.gov/~hoschek/colt/`.

[CEvA07] Mani Chandy, Opher Etzion, and Rainer von Ammon, editors. *Event Processing*, volume 07191 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. `http://drops.dagstuhl.de/portals/index.php?semnr=07191`.

[CKAK94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*, pages 606–617. Morgan Kaufmann, 1994.

[CL04] Jan Carlson and Björn Lisper. An event detection algebra for reactive systems. In *Proc. ACM Int. Conf. On Embedded Software*, pages 147–154. ACM, 2004.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, World Wide Web Consortium, 1999.

[Com] Commons-Collections. `http://larvalabs.com/collections`.

[Con07] Dan Connolly. Gleaning resource descriptions from dialects of languages (GRDDL). W3C recommendation, World Wide Web Consortium, 2007.

[Cor07] Coral8, Inc. Complex Event Processing: Ten design patterns. White Paper. `http://www.coral8.com/system/files/assets/pdf/Coral8DesignPatterns.pdf`, 2007.

[Coş07] Fatih Coşkun. Pattern-based updates for the Web: Refinement of syntax and semantics in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2007.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[Cri03] M. Crispin. Interactive mail access protocol — version 4rev1. RFC 3501, The Internet Society, 2003.

[CT04] John Cowan and Richard Tobin. XML information set (second edition). W3C recommendation, World Wide Web Consortium, 2004.

[dB08] Jos de Bruijn. RIF RDF and OWL compatibility. W3C working draft, World Wide Web Consortium, 2008.

[DFFT02] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of xml documents. In *Proc. Int. Conf. on Data Engineering*, pages 341–344. IEEE Computer Society, 2002.

[DIR07] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 1–8. ACM, 1984.

[DMO01] Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink) version 1.0. W3C recommendation, World Wide Web Consortium, 2001.

[DS99a] A. Daneels and W. Salter. What is SCADA? In *Proc. Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, pages 339–343. Comitato Conferenze ELETTRA, 1999.

[DS99b] Martin Duerst and Michel Suignard. Internationalized resource identifiers (IRIs). RFC 3987, The Internet Society, 1999.

[DW07] Wlodzimierz Drabent and Artur Wilk. Extending xml query language xcerpt by ontology queries. In *Int. Conf. on Web Intelligence*, pages 447–451. IEEE Computer Society, 2007.

[EB06] AnnMarie Ericsson and Mikael Berndtsson. Detecting design errors in composite events for event triggered real-time systems using timed automata. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, pages 39–50. IEEE Computer Society, 2006.

[ebX] Electronic business using eXtensible Markup Language. `http://www.ebxml.org`.

[Eck05] Michael Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2005.

[Ecl] Eclipse Foundation. Eclipse — an open development platform. `http://www.eclipse.org`.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL:2003 has been published. *SIGMOD Record*, 33(1):119–126, 2004.

[EN03] Eiman Elnahrawy and Badri Nath. Cleaning and querying noisy sensors. In *Proc. ACM Conf. on Wireless Sensor Networks and Applications*, pages 78–87. ACM, 2003.

[EPBS07] AnnMarie Ericsson, Paul Pettersson, Mikael Berndtsson, and Marco Seiriö. Seamless formal verification of Complex Event Processing applications. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 50–61. ACM, 2007.

[Esp] EsperTech Inc. Event stream intelligence: Esper & NEsper. `http://esper.codehaus.org`.

[Est08] Olga Estekhina. Well-founded semantics and local stratification for Xcerpt programs. Project thesis (Projektarbeit), Institute for Informatics, University of Munich, 2008.

[Etz05] Opher Etzion. Towards an event-driven architecture: An infrastructure for event processing (position paper). In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 1–7. Springer, 2005.

[F+99] R. Fielding et al. Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, 1999.

[FGV05] Michael Fisher, Dov Gabbay, and Lluis Vila, editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[FHH04] Richard Fikes, Patrick J. Hayes, and Ian Horrocks. OWL-QL – a language for deductive query answering on the Semantic Web. *J. Web Semantics*, 2(1):19–29, 2004.

[FLB+06] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF querying: Language constructs and evaluation methods compared. In *Reasoning Web, Int. Summer School*, volume 4126 of *LNCS*, pages 1–52. Springer, 2006.

[FM77] Charles Forgy and John P. McDermott. OPS, a domain-independent production system language. In *Proc. Int. Joint Conference on Artificial Intelligence*, pages 933–939. William Kaufmann, 1977.

[For81] Charles Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[For82] Charles L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intelligence*, 19(1):17–37, 1982.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fur03] Tim Furche. Optimizing multiple queries against XML streams. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2003.

[Fur08] Tim Furche. *Implementation of Web Query Languages Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost.* PhD thesis, Institute for Informatics, University of Munich, 2008.

[G+03] Martin Gudgin et al. SOAP 1.2. W3C recommendation, World Wide Web Consortium, 2003.

[GA02] Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 2453 of *LNCS*, pages 547–556. Springer, 2002.

[GA04] Rodolfo Gómez and Juan Carlos Augusto. Durative events in active databases. In *Proc. Int. Conf. on Enterprise Information Systems*, pages 306–311, 2004.

[GAC06] Vihang Garg, Raman Adaikkalavan, and Sharma Chakravarthy. Extensions to stream processing architecture for supporting event processing. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 4080 of *LNCS*, pages 945–955. Springer, 2006.

[GAE06] Thanaa M. Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Record*, 35(1):3–8, 2006.

[GD93] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*, pages 23–39. Springer, 1993.

[GD94] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proc. Int. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9. IEEE, 1994.

[GDP+06] Torsten Greiner, Willy Düster, Francis Pouatcha, Rainer von Ammon, Hans-Martin Brandl, and David Guschakowski. Business activity monitoring of norisbank taking the example of the application easyCredit and the future adoption of Complex Event Processing (CEP). In *Proc. Int. Symp. on Principles and Practice of Programming in Java*, pages 237–242. ACM, 2006.

[GGM+04] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJS92a] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 327–338. Morgan Kaufmann, 1992.

[GJS92b] Narain H. Gehani, H.V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 81–90. ACM, 1992.

[GJS93] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Compose: A system for composite specification and detection. In *Advanced Database Systems*, volume 759 of *LNCS*, pages 3–15. Springer, 1993.

[GL92] Benoit A. Gennart and David C. Luckham. Validating discrete event simulations using event pattern mappings. In *Proc. Design Automation Conference*, pages 414–419. IEEE Computer Society, 1992.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 328–339. ACM, 1995.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. Int. Conf. on Data Engineering*, pages 209–218. IEEE Computer Society, 1993.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Gra95] Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[Gra06] Hendrik Grallert. Propagation of updates in distributed web data: A use case for the language XChange. Project thesis, Institute for Informatics, University of Munich, 2006.

[GS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CUM-CS-94-166, Carnegie Mellon University, 1994.

[GUW01] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[H+08] Arnaud Le Hors et al. Document object model (DOM) level 3 core specification. W3C recommendation, World Wide Web Consortium, 2008.

[Hav05] Michael Havey. *Essential Business Process Modeling*. O'Reilly, 2005.

[Hay04] Patrick Hayes. RDF semantics. W3C recommendation, World Wide Web Consortium, 2004.

[HBBM96] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using events to build large scale distributed applications. In *Proc. ACM SIGOPS European Workshop on Systems Support for Worldwide Applications*, pages 9–16. ACM, 1996.

[HV02] Annika Hinze and Agnès Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, pages 61–65. IEEE, 2002.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[IBM04] IBM. Common Base Event. `http://www.ibm.com/developerworks/webservices/library/ws-cbe`, 2004.

[ILO] ILOG. ILOG JRules. `http://www.ilog.com/products/jrules`.

[JAF+06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative support for sensor data cleaning. In *Proc. Int. Conf. on Pervasive Computing*, volume 3968 of *LNCS*, pages 83–100. Springer, 2006.

[JBo] JBoss.org. Drools. `http://www.jboss.org/drools`.

[JCG+92] Christian S. Jensen, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3):35–43, 1992.

[JP06] Daniel Jobst and Gerald Preissler. Mapping clouds of SOA- and business-related events for an enterprise cockpit in a Java-based environment. In *Proc. Int. Symp. on Principles and Practice of Programming in Java*, pages 230–236. ACM, 2006.

[JUN] JUNG — Java Universal Network/Graph Framework. `http://jung.sourceforge.net/`.

[Kay07] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, World Wide Web Consortium, 2007.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C recommendation, World Wide Web Consortium, 2004.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(2):323–350, 1977.

[KLG07] Martin Kersten, Erietta Liarou, and Romulo Goncalves. A query language for a data refinery cell. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[Kow92] Robert A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12(1&2):121–146, 1992.

[KS86] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Compututing*, 4(1):67–95, 1986.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LC86a] Tobin J. Lehman and Michael J. Carey. Query processing in main memory database management systems. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 239–250. ACM, 1986.

[LC86b] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proc. Int. Conf. on Very Large Databases*, pages 294–303. Morgan Kaufmann, 1986.

[Llo93] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

[LS08] David Luckham and Roy Schulte. Event processing glossary. `http://complexevents.com/?p=361`, May 2008.

[Luc02] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[Luc08] David C. Luckham. A short history of Complex Event Processing. part 1: Beginnings. `http://complexevents.com/?p=321`, 2008.

[LVB+93] David C. Luckham, James Vera, Doug Bryan, Larry M. Augustin, and Frank C. Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, 1993.

[MAA05a] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the Semantic Web: Dealing with language heterogeneity. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 30–44. Springer, 2005.

[MAA05b] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Ontology- and resources-based approach to evolution and reactivity in the Semantic Web. In *Proc. Int. Conf. on Ontologies, Databases, and Applications of Semantics*, volume 3761 of *LNCS*, pages 1553–1570. Springer, 2005.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[MBM08] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. A framework for performance evaluation of Complex Event Processing systems. In *Proc. Int. Conf. on Distributed Event-Based Systems, Demonstrations*, pages 313–316. ACM, 2008.

[McC02] David W. McCoy. Business activity monitoring: Calm before the storm. Technical Report LE-15-9727, Gartner, Inc., 2002. `http://www3.gartner.com/resources/105500/105562/105562.pdf`.

[McD86] Drew McDermott. Tarskian semantics, or no notation without denotation! In *Readings in Natural Language Processing*, pages 167–169. Morgan Kaufmann, 1986.

[ME01] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1):1–10, 2001.

[MFP06] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[MH69] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[Mir87] Daniel P. Miranker. TREAT: A better match algorithm for AI production system matching. In *Proc. AAAI Natl. Conf. on Artificial Intelligence*, pages 42–47. AAAI Press, 1987.

[MKW+02] Highland Mary Mountain, Jacek Kopecky, Stuart Williams, Glen Daniels, and Noah Mendelsohn. SOAP version 1.2 email binding. W3C note, World Wide Web Consortium, 2002.

[MM04] Frank Manola and Eric Miller. RDF primer. W3C recommendation, World Wide Web Consortium, 2004.

[MR96] J. Myers and M. Rose. Post office protocol - version 3. RFC 1939, The Internet Society, 1996.

[MS] MS Analog Software. ruleCore(R) Complex Event Processing (CEP) Server. `http://www.rulecore.com`.

[MSS97] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[MV07] John Morrell and Stevan D. Vidich. Complex Event Processing with Coral8. White Paper. `http://www.coral8.com/system/files/assets/pdf/Complex_Event_Processing_with_Coral8.pdf`, 2007.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[MZ97] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 440–451. ACM Press, 1997.

[NEX] NEXMark Benchmark. `http://datalab.cs.pdx.edu/niagara/NEXMark/`.

[NMMK07] Fred Niederman, Richard G. Mathieu, Roger Morley, and Ik-Whan Kwon. Examining RFID applications in supply chain management. *Communications of the ACM*, 50(7):92–101, 2007.

[Ora] Oracle Inc. Complex Event Processing in the real world. White Paper. `http://www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf`.

[P+06] Paula-Lavinia Patranjan et al., editors. *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*. Springer, 2006.

[Par07] Terence Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Programmers, 2007.

[Pas08] Adrian Paschke. Design patterns for complex event processing. In *Proc. Int. Conf. on Distributed Event-Based Systems, Fast Abstracts*, 2008. http://debs08.dis.uniroma1.it/pdf/fa-paschke.pdf.

[Pat98] Norman W. Paton, editor. *Active Rules in Database Systems*. Springer, 1998.

[Păt05] Paula-Lavinia Pătrânjan. *The Language XChange: A Declarative Approach to Reactivity on the Web*. PhD thesis, Institute for Informatics, University of Munich, 2005.

[Pix08] Tom Pixley. Document object model (DOM) level 2 events specification. W3C recommendation, World Wide Web Consortium, 2008.

[PKB+07] Adrian Paschke, Alexander Kozlenkov, Harold Boley, Said Tabet, Michael Kifer, and Mike Dean. Reaction RuleML. `http://ibis.in.tum.de/research/ReactionRuleML/`, 2007.

[PS06] Adrian Paschke and Elisabeth Schnappinger-Gerull. A categorization scheme for SLA metrics. In *Proc. Conf. on Service Oriented Electronic Commerce*, volume 80 of *LNI*, pages 25–40. GI, 2006.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, World Wide Web Consortium, 2008.

[PvA08] Adrian Paschke and Rainer von Ammon. EuroPLoP 2008 focus group domain-specific complex event and rule patterns. `http://www.citt-online.de/downloads/EuroPLoP_CEP_Focus.pdf`; see also `http://hillside.net/europlop/`, 2008.

[PvdH07] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: Approaches, rechnologies and research issues. *The VLDB Journal*, 16(3), 2007.

[REW] REWERSE Network of Excellence (Funded in the Sixth Framework Programme of the European Union 2004–2008). Working group I5 — evolution and reactivity. `http://www.rewerse.net/i5`.

[RGR08] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. Satisfying complex data needs using pull-based online monitoring of volatile data sources. In *Proc. Int. Conf. on Data Engineering*, pages 1465–1467. IEEE, 2008.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C recommendation, World Wide Web Consortium, 1999.

[RIF] RIF WG. Rule interchange format working group charter. `http://www.w3.org/2005/rules/wg/charter`.

[RLBS08] Christopher Ré, Julie Letchner, Magdalena Balazinska, and Dan Suciu. Event queries on correlated probabilistic streams. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 715–728. ACM, 2008.

[Rom06] Inna Romanenko. Use cases for reactivity on the Web: Using ECA rules for business process modeling. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2006.

[Ron97] Claudia Roncancio. Toward duration-based, constrained and dynamic event types. In *Proc. Int. Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *LNCS*, pages 176–193. Springer, 1997.

[Ros06] Riccardo Rosati. Integrating ontologies and rules: Semantic and computational issues. In *Reasoning Web, Int. Summer School*, volume 4126 of *LNCS*, pages 128–151. Springer, 2006.

[San] Sandia National Laboratories. Jess, the rule engine for the Java(TM) platform. http://herzberg.ca.sandia.gov/.

[SB04] Sebastian Schaffert and François Bry. Querying the Web reconsidered: A practical introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.

[SB05] Marco Seiriö and Mikael Berndtsson. Design and implementation of an eca rule markup language. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 98–112. Springer, 2005.

[Sch03] Roy W. Schulte. The growing role of events in enterprise applications. Technical Report AV-20-3900, Gartner, Inc., 2003. http://www.gartner.com/resources/116100/116129/116129.pdf.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* PhD thesis, Institute for Informatics, University of Munich, 2004.

[Sch07] Stefanie Scherzinger. Bulk data in main memory-based xquery evaluation. In *Proc. Int. Workshop on XQuery Implementation, Experience and Perspectives*, 2007.

[Sch08] Scarlet Schwiderski-Grosche. Spatio-temporal reasoning with composite events in mobile systems. In *Proc. Int. Conf. on Distributed Event-Based Systems, Fast Abstracts*, 2008. http://debs08.dis.uniroma1.it/pdf/fa-grosche.pdf.

[Sed90] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. Int. Conf. on Very Large Databases*, pages 510–521. Morgan Kaufmann, 1994.

[SM06] Kelly Sims and Tineke Mertens. Heineken, IBM, Safmarine and University of Amsterdam launch wireless "beer living lab". IBM Press Release, http://www-03.ibm.com/press/us/en/pressrelease/20514.wss, 2006.

[SR90] Young-Chul Shim and C. V. Ramamoorthy. Monitoring and control of distributed systems. In *Proc. Int. Conf. on Systems Integration*, pages 672–681. IEEE Computer Society, 1990.

[SSS+03] César Sánchez, Sriram Sankaranarayanan, Henny Sipma, Ting Zhang, David L. Dill, and Zohar Manna. Event correlation: Language and semantics. In *Proc. Int. Conf. on Embedded Software*, volume 2855 of *LNCS*, pages 323–339. Springer, 2003.

[SSSM05] César Sánchez, Matteo Slanina, Henny B. Sipma, and Zohar Manna. Expressive completeness of an event-pattern reactive programming language. In *Int. Conf. on Formal Techniques for Networked and Distributed Systems*, volume 3731 of *LNCS*, pages 529–532. Springer, 2005.

[Sub] Subversion. http://subversion.tigris.org/.

[Sun] Sun Microsystems, Inc. Java(TM) platform, standard edition 6. http://java.sun.com/javase/6/.

[Sun06] Sun Microsystems, Inc. Java(TM) platform, standard edition 6 API specification. http://java.sun.com/javase/6/docs/api/, 2006.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language guide. W3C recommendation, World Wide Web Consortium, 2004.

[SZZA01] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *Proc. ACM Symp. on Principles of Database Systems*, pages 71–81. ACM, 2001.

[SZZA04] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems*, 29(2):282–318, 2004.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, second edition, 1999.

[TPC] TPC. Transaction Processing Performance Council. `http://www.tpc.org/`.

[TS86] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *Proc. Int. Conf. on Logic Programming*, volume 225 of *LNCS*, pages 84–98. Springer, 1986.

[VH07] Gottfried Vossen and Stephan Hagemann. *Unleashing Web 2.0: From Concepts to Creativity.* Morgan Kaufmann, 2007.

[Vie86] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Proc. Int. Conf. on Expert Database Systems*, pages 253–267. Benjamin Cummings, 1986.

[War92] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[WBG08] Karen Walzer, Tino Breddin, and Matthias Groch. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 147–155. ACM, 2008.

[WC96] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance Complex Event Processing over streams. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 407–418. ACM, 2006.

[WGET08] Segev Wasserkrug, Avigdor Gal, Opher Etzion, and Yulia Turchin. Complex event processing over uncertain data. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 253–264. ACM, 2008.

[WRGD07] Walker M. White, Mirek Riedewald, Johannes Gehrke, and Alan J. Demers. What is “next” in event processing? In *Proc. ACM Symp. on Principles of Database Systems*, pages 263–272. ACM, 2007.

[WTV+07] Georg Wittenburg, Kirsten Terfloth, Freddy López Villafuerte, Tomasz Naumowicz, Hartmut Ritter, and Jochen H. Schiller. Fence monitoring - experimental evaluation of a use case for wireless sensor networks. In *Proc. Europ. Conf. on Wireless Sensor Networks*, volume 4373 of *LNCS*, pages 163–178. Springer, 2007.

[WvASW07] Alexander Widder, Rainer von Ammon, Philippe Schaeffer, and Christian Wolff. Identification of suspicious, unknown event patterns in an event cloud. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 164–170. ACM, 2007.

[Xce] Xcerpt. `http://xcerpt.org`.

[ZS01] Dong Zhu and Adarshpal S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*, pages 586–589. IEEE, 2001.

[ZU96] Robert J. Zhang and Elizabeth A. Unger. Event specification and detection. Technical Report TR CS-96-8, Kansas State University, 1996. `http://citeseer.ist.psu.edu/zhang96event.html`.

[ZU99] Detlef Zimmer and Rainer Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*, pages 392–399. IEEE, 1999.

About the Author

Michael Eckert studied computer science with a minor in mathematics at University of Munich (LMU, 1999-2005) and at University of Washington (2002-2003). He received his M.Sc. equivalent with distinction ("Diplom-Informatiker mit Auszeichnung") from LMU in May 2005. Since July 2005, he is working as a research and teaching assistant at the Institute for Informatics of LMU in the programming and modeling languages group of Prof. François Bry. His research interests include Complex Event Processing, reactive Web systems and reactive rule languages, update languages for Web data, and Web and database technology in general.