Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

An Approach to Complex Event Detection in the Web

Gastón E. Tagni

Computational Logic Masters Thesis

Dissertação apresentada na Faculdade de Ciencias e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Lógica Computacional.

Orientador: Prof. José Júlio Alferes

Lisboa
(2007)

Acknowledgements

First of all, I would like to express my gratitude to my thesis supervisor *Prof. José Júlio Alferes*, of the *New University of Lisbon*, for guiding me through the entire research process.

I would also like to thank to the professors of CENTRIA for the assistance they provided me during my year at UNL, specially to *Prof. Luís Moniz Pereira*, *Prof. Carlos Damásio*, *Prof. João A. Leite* and *Prof. João Moura Pires* for everything they taught me. I am also thankful for the support provided by *Prof. Enrico Franconi* of the *Free University of Bozen-Bolzano* during the first year of the masters.

I gratefully acknowledge the financial support given by the *European Commission* through the *Erasmus Mundus Program*, which made possible the completion of my studies at the *Free University of Bolzano* and the *New University of Lisbon*.

I would like to say thanks to my friends at UNIBZ, especially to *Luciana*, *Magdalena* and *Juan* for their support and those wonderful moments we spent together. I also thank my friends and colleagues at UNL, in particular *Dejan*, *Jignesh*, *Freddy* and *Vivek* for their unconditional and valuable help and for sharing with me great moments.

I am especially grateful to my parents and brother for their constant support and for being with me all the time, even at the distance.

I also thank *Prof. Laura Cecchi* and *Prof. Pablo Filottrani* from Argentina, for everything they taught me and for giving me the opportunity to pursue my academic goals.

I would also like to say thanks to *Ricardo Amador* of CENTRIA, whose help and suggestions during the implementation phase of the system were very important.

Last but not least, I thank Andrea. This thesis could not have been written in English without her help.

Summary

Reactivity, *the ability to detect events and react to them accordingly by executing appropriate actions*, has long been studied in the field of Active Databases and more recently, in the context of the World Wide Web.

A common approach for implementing reactive behaviour and evolution in the Web is the use of reactive languages. Such languages use reactive rules for describing the reactive behaviour of a system and in particular, *Event-Condition-Action (ECA) rules*. The declarative semantics of such rules specifies that upon event detection, if the system's state satisfies the stated conditions then a group of actions are executed.

One of the key aspects in the implementation of reactivity is thus the development of event detectors. These systems are responsible for registering event expressions, detecting the events specified by registered expressions and then notifying the interested systems about the occurrence of events. Events can be broadly classified into *atomic* and *composite events*. An atomic event is something that happens atomically (it occurs completely or not at all) at a given point in time. On the other hand, composite events are combinations of other events (atomic or composite). They represent situations that occur when their constituent events occur and they are expressed by expressions of an event algebra.

In this work, we present the implementation of a complex event detector for the Web. It detects composite events specified by expressions of an illustrative sublanguage of the event algebra SNOOP. This notification system was implemented as a component of the *General Framework for Reactivity and Evolution in the Web* [7], proposed by the *Evolution and Reactivity Working Group* of the REWERSE [1] project. In addition, we present a comparative analysis of some active languages and more specifically, of the event detectors used by them. The aim of this analysis was to compare different approaches for implementing such systems.

[1] `http://rewerse.net`

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Contributions of this Work . . . 4
- 1.3 Outline of the Thesis . . . 5

2 Reactivity and Evolution in the Web **7**
- 2.1 Introduction . . . 8
  - 2.1.1 Specifying Reactive Behaviour: ECA Paradigm . . . 8
  - 2.1.2 Reactivity in the (Semantic) Web . . . 9
- 2.2 General Aspects of Active Rules . . . 11
- 2.3 Related Work . . . 13
  - 2.3.1 Research on Active Rules . . . 13
  - 2.3.2 Update and Query Languages . . . 14

3 Event Detection: General Aspects and Techniques **15**
- 3.1 Event Model . . . 16
- 3.2 General Aspects of Event Detection . . . 18
- 3.3 Techniques for Event Detection . . . 21
  - 3.3.1 Detection of Primitive Events . . . 21
  - 3.3.2 Detection of Primitive Events in XML . . . 22
  - 3.3.3 Techniques for Composite Event Detection . . . 25
  - 3.3.4 Composite Event Detection in XML . . . 29
- 3.4 Related Works . . . 31
  - 3.4.1 Event Detection in Active Databases . . . 31
  - 3.4.2 Research on Event Languages . . . 31
  - 3.4.3 Event Detection in XML and RDF . . . 32
  - 3.4.4 Event Notification Systems . . . 32

4 Active Languages for the (Semantic) Web **33**
- 4.1 RDFTL: A Trigger Language for RDF . . . 34
  - 4.1.1 Definition of the Language . . . 34
- 4.2 Active XQuery Language . . . 36
  - 4.2.1 Syntax of the language . . . 37

4.2.2 Underlying Update Model and Language . . . 39
4.3 XChange Active Language . . . 40
4.3.1 Event Model and Event Messages . . . 41
4.3.2 Event Queries . . . 42
4.4 A General Language for Reactivity in the Semantic Web . . . 47
4.4.1 ECA Rules . . . 48
4.4.2 Rule Components and Languages . . . 48
4.4.3 Interaction among Rule Components. Logical Variables . . . 50

5 Comparative Framework 53
5.1 Definition of the Comparative Framework . . . 54
5.2 RDFTL: RDF Triggering Language . . . 55
5.2.1 Supporting RDFTL ECA Rules in Distributed Environments . . . 56
5.2.2 Supporting RDFTL ECA Rules in Centralized Environments . . . 59
5.3 Active XQuery . . . 60
5.3.1 Semantics and Execution Model . . . 60
5.4 XChange Active Language . . . 64
5.4.1 System Architecture . . . 64
5.4.2 Event Detection in XChange . . . 66
5.4.3 Working with Logical Variables . . . 68
5.5 Evaluation Results . . . 69
5.5.1 RDFTL . . . 69
5.5.2 Active XQuery . . . 72
5.5.3 XChange . . . 75

6 Implementation of a Complex Event Detector 79
6.1 Introduction . . . 80
6.2 Architecture for Implementing the ECA Framework . . . 80
6.3 Implementation Aspects . . . 82
6.3.1 Architecture of the Event Detector . . . 82
6.3.2 Event Model and Event Expressions . . . 84
6.3.3 Markup Language for ECA Rules . . . 85
6.3.4 Representing Event Expressions: Event Trees and Event Graph . 86
6.3.5 Registration of Event Expressions . . . 87
6.3.6 Composite Event Detection . . . 88
6.3.7 Deletion of Registered Event Expressions . . . 92
6.4 Evaluation of the Implemented System . . . 93

7 Conclusion 99
7.1 Design and Implementation of Event Detectors for the Web . . . 100
7.1.1 Event Languages . . . 100

7.1.2 Event Detection Semantics 101
7.1.3 Techniques for Detecting Events 101
7.1.4 Representation of Events 103
7.1.5 Extracting Information from Events 104
7.1.6 Communication of Events 105
7.2 Future Work 105

A Class Diagram **115**

List of Figures

3.1 Ontology of atomic and composite events. 17
3.2 A possible classification of atomic events. 17
3.3 Two versions of an XML document. The older version is on the left-hand side 24
3.4 Combination of two event trees. 27
3.5 Combination of two Petri nets 28

4.1 Path expressions grammar 34
4.2 Triggers' syntax in Active XQuery 38
4.3 Definition of an Active XQuery trigger 39
4.4 Definition of an Active XQuery trigger 40
4.5 Event message containing information about the availability of a new book. 42
4.6 Atomic event query to detect the event message defined in the previous example. 43
4.7 Using Event Queries and Event Messages in event detection. 44
4.8 Rules an rule components. 49
4.9 Example of an ECA rule defined in the language. 51

5.1 System Architecture for Distributed Environments 56
5.2 System Architecture for Centralized Environments 59
5.3 XML tree showing the result of executing a "bulk" insert 62
5.4 Architecture of a Active XQuery Rule System 64
5.5 Architecture of a prototype Rule Engine for XChange active rules 65
5.6 A composite event query defined using the conjunctive operator. 67
5.7 Event query represented using an event tree 68
5.8 A composite event query defined using the conjunctive operator. 69

6.1 An architecture supporting the proposed ECA Framework 81
6.2 The architecture of the event detector 84
6.3 A composite event expression. 86
6.4 An Event Graph sharing two Event Trees 87
6.5 Event graph before and after deleting expression B 93

A.1 Class Diagram 116

Chapter 1

Introduction

The actual *World Wide Web* can be seen as a large collection of information distributed across different data sources. These data sources store the information using text formats such as XML and HTML, and provide the users with mechanisms to retrieve and access the data. In this sense, the Web is nowadays mostly a passive data repository, providing no or very limited mechanisms for reactivity and automatic evolution of information.

The development of advanced distributed web-based applications such as e-commerce and business-to-business applications has suggested the need for techniques that support reactive behaviour. That is, the ability to react to events occurring throughout the web and act automatically by executing appropriate actions. Currently, the majority of web-based applications supporting reactivity implement a limited form of reactive behaviour. These systems are able to detect changes that occur locally at the nodes where the data is stored, for example by using triggers in local databases. Although this does provide a simple level of reactivity, a more global mechanism is needed in order to give applications the ability to react to non-local events, that is, events that occur at some other points in the web.

One of the key aspects for supporting reactivity and evolution in the Web is the design and use of reactive languages [7,10,58,63]. Based on the *Event-Condition-Action* (ECA) paradigm, these languages use active rules to implement reactive behaviour. ECA rules are a simple but powerful mechanism to define reactivity in a declarative manner. An ECA rule specifies that after the occurrence of an event, a set of actions must be executed; provided that stated conditions hold. This execution model suggests the need for the implementation of event detection engines, capable of detecting events not only locally but also globally in the Web.

1.1 Motivation

The *General Framework for Reactivity and Evolution in the Web* [7], proposed by the *Evolution and Reactivity Working Group* of the REWERSE [1] project, is an ontology-based approach for describing (reactive) behaviour and evolution in the Web that follows the ECA paradigm. As usual, ECA rules are defined by specifying their event, condition and action components. However, and in contrast with other approaches to reactivity, the framework allows for the composition of different (sub)languages for the specification of each of these components; thus, the framework is able to deal with the heterogeneity of languages and concepts found in the Semantic Web. For example, two ECA rules stored in an ECA rule base might be defined by using different event languages in order to specify their event components. In other words, the framework does not fix the (sub)languages used for specifying the components of ECA rules.

The ECA framework is being implemented using a service-based, distributed architecture [7, 54] which associates every language with a Web Service that implements its semantics. For example, a Web Service could implement the semantics of an event algebra and hence provide composite event detection capabilities to other systems. The architecture consists of a set of modules, each of them implementing a particular service such as *event detection*, *query evaluation* and *action execution*, among others. Not all the services belong to the ECA framework, in the sense that other modules might exist that implement a particular service. From the architecture's point of view, these existing services that are integrated into the framework are called *opaque services*. This heterogeneity of services is handled by the framework by making no assumptions on the type of services that can be integrated with the architecture. However, in order for the services to be able to interact and cooperate they must implement the communication by means of variables bindings. In other words, services communicates with each other by means of variable bindings; i.e. by interchanging *variable-value pairs* that act as input and output variables to the services. The implementation of this type of communication may require the use of appropriate *framework-aware wrappers* to help bridge the gap between the functionality of the framework's services and that of opaque services.

The prototype is being implemented incrementally. Consequently, in early stages of the implementation some of the components may need to be simulated by "dummy" modules. At the moment of writing this thesis, the architecture consists of the following services or modules:

- *ECA Rule Engine.* This represents the most important component of the architecture as it is in charge of handling ECA rules. The module's functionality comprises registration of ECA rules, identification of the rule's components, registration of the event part using appropriate event detectors, processing of variable bindings (answers from other services and input data for invoking other

[1] http://rewerse.net

services), evaluation of query and condition parts and execution of actions by invoking the appropriate action processors.

An ECA engine processes incoming rule registration requests and evaluates the event part by invoking the appropriate event detector (based on the event language used for specifying the event). After an event is detected, the ECA engine processes the answer returned by the event detector (an event instance) and proceeds with the evaluation of the condition part, which may include the evaluation of a query and the test of a boolean condition. After the condition part is successfully evaluated, the ECA engine invoke a service to execute the actions in the action part of a rule. A first version of a prototype is presented in [65].

- *Event Detector.* Two types of event detectors are required for implementing an ECA framework for the Web. First, *atomic event detectors* are used for detecting atomic events. This type of events occurs at the database level. Second, *composite event detectors* implementing an event algebra are used for detecting composite or complex events in the Web. In order to detect composite events, a composite event detector must be able to receive and process atomic events. The communication between atomic and composite event engines can be implemented using different approaches: *straightforward*, *application-centered* and *language-centered* [54].

- *Query Engines.* In general lines, a *Query Processor* is responsible for the evaluation of queries specified in the condition part of ECA rules. Here, existing query services can be integrated into the architecture as opaque services. For example, an XML query engine could be used for evaluating queries against XML repositories. A query engine evaluates query expressions, which can be specified using one of different query languages (such as XQuery [20] or Xcerpt [66] or, alternatively, they can be specified using the query language of the node the query refers to).

- *Action Engines.* This component is responsible for executing actions specified in the action part of ECA rules. An action engine implements an action language (e.g., we can use Prova [50] as an action language); which is used for specifying actions. As in the case of events, actions can be classified into *atomic actions*, *generic actions* and *composite actions*. Atomic actions occur at the database level and correspond to update operations expressed using an update language. Once again, existing engines implementing update languages can be reused. Generic actions are for example actions that send messages to other modules. Finally, composite actions are a combination of the previous ones. This classification calls for the implementation of different action engines.

- *Event Brokers.* Event brokers are responsible for detecting atomic events of a par-

ticular domain. For example, a composite event detector may need to be notified about atomic events occurring at an online marketplace, e.g. insertion of new products in the database. In this case, an event detector that is able to detect update operations at the database level would provide the required atomic events. In general, event brokers will be reused as opaque services as they will be developed for particular domains.

The main motivation for our work is the lack of an event detector that can be used in the context of the ECA framework described before. At the moment of writing this thesis, the framework simulates the event detection service using a "dummy" module. Although this is enough for testing and implementing the basic operations of the ECA engine, a more powerful event engine is desired. With this in mind, we proposed ourselves to developed a complex event detector for the Web that could be, at the same time, integrated into this framework.

1.2 Contributions of this Work

When implementing an event detector for the Web several issues must be considered. First, the type of events that we want to detect is important as this determines the strategy to be used for detecting them. In addition, the type of events (the ontology of events) plays an important role in the selection of the event algebra to be used in order to specify composite events. Second, different strategies for detecting a particular type of event might exist and thus, an analysis considering the advantages and disadvantages of each of them should be made. Third, several methods for computing event data (data communicated by events) may exist. Another issues are the communication of atomic events between different systems, the notification of composite events to interested systems and the registration of event expressions among others.

This in turn, prompted us to investigate different existing approaches to event detection in the context of the Web. More specifically, we decided to study a number of active languages by analyzing the techniques they use for implementing reactive functionality. In particular, we were interested in the implementation aspects of the event detectors used by these languages. The languages considered in this thesis and the reasons for selecting them are outlined below.

- *RDF Triggering Language* The language proposed in [58] is specially interesting because it allows the implementation of reactive behaviour on RDF repositories. As such, it is focused on detecting events that reflect changes in the nodes of an RDF graph.

- *Active XQuery* This language [10] provides the means for implementing reactive functionality on XML repositories. Due to the fact XML has become a key technology for representing and exchanging information among systems in the Web,

we decided to analyze the implementation of reactive functionality in this context. As in the case of RDF, events in XML reflect changes in the content of XML documents.

- *XChange Active Language* Compared with the previous two languages, XChange [63] is a generic, declarative reactive language for detecting events that occur in the Web.

In summary, the contributions of this thesis are the following:

- *Comparative analysis* The comparative analysis of different active languages constitutes the starting point of the design and implementation of our event detector. This analysis helped us to identify the key problems that arise when implementing event detectors as well as the alternatives for solving them. Moreover, the results of the comparison can be used as guidelines for the development of such notification systems.

- *Implementation of a Complex Event Detector* The main contribution of our work is the implementation of a complex event detector for the Web. Such notification system was designed to work as part of the ECA framework mentioned before. However, it can be easily integrated into existing systems as long as the communication mechanism is implemented by means of logical variables (variable bindings). This event detector detects composite events specified by expressions of an illustrative sublanguage of the event algebra SNOOP.

1.3 Outline of the Thesis

The thesis is organized as follows. Chapter 1 (this chapter) introduces the topic of the thesis, presents the motivations for this work and the goals that we try to achieve. Then, in Chapter 2 we introduce the reader to the concepts of Reactivity and Evolution, specially in the context of the (Semantic) Web. We describe general aspects of Reactivity and present the related works. Chapter 3 reports on the general aspects of event detection. We explore several approaches to event detection in different scenarios and present the related works. After this, in Chapter 4 we proceed to the presentation of the Reactive Languages analyzed in this work. Then, Chapter 5 we evaluate the Active Languages by focusing our attention on the event detection capabilities provided by the languages processors. Chapter 6 presents the implementation of a complex event engine that detects events in the context of the *General Language for Reactivity and Evolution in the Web* [7]. Finally, Chapter 7 presents the conclusions and discusses future works.

Chapter 2

Reactivity and Evolution in the Web

Contents

2.1 Introduction 8
2.2 General Aspects of Active Rules 11
2.3 Related Work 13

In this chapter we discuss the concepts of Reactivity and Evolution in the context of the (Semantic) Web and introduce the ECA paradigm, which has been used for implementing reactive behaviour in several domains. Then, we mention general aspects of reactive rules and finally, we describe some related works.

2.1 Introduction

Reactivity, *the ability to react to events that occur at a certain point in time and place and, act accordingly to them by executing actions*, has long been considered in the field of *Active database systems* [61, 72]. Traditional databases are "passive" data repositories that do not provide means for reacting to changes in the state of the database. As a consequence of this, external applications are responsible for detecting changes in the database and reacting accordingly to them by executing appropriate actions. In other words, reactive behaviour must be implemented outside the DBMS. For example, consider a book store catalog implemented using a database that stores information about books and authors. In this scenario, whenever a book is deleted from the `book` table, the author's information should also be deleted from the corresponding `author` table. In traditional databases this behaviour is implemented in the application programs accessing the database. Moreover, if users want to be notified about the insertion of a new book, every application program using the database must include code so as to detect such situation and notify the users.

In Active database systems, reactive behaviour is implemented inside the DBMS. As pointed out in [62], an Active DBMS extends a "passive" DBMS with both a knowledge and execution model so as to support reactive behaviour. The knowledge model enables the description or specification of relevant situations to which the DBMS should react and the actions to be executed in response to those situations. Additionally, the knowledge model may allow the specification of conditions that must hold when certain situations occur. The execution model instead defines how reactive behaviour is actually implemented in the Active DBMS. In the previous example, an active database would implement the reactive behaviour inside the DBMS and thus, application programs do not need to implement it. Furthermore, any application program accessing the database would benefit from it. Active Databases demonstrated to be very useful for a wide range of applications. They have been used for implementing active views (maintenance of derived data), for enforcing integrity constraints and for the implementation of alerters.

2.1.1 Specifying Reactive Behaviour: ECA Paradigm

A common approach for implementing reactive behaviour is the use of *ECA rules* [1]. *Event-Condition-Action* rules (ECA rules) are a simple but powerful paradigm for specifying reactive behaviour in a declarative manner. An ECA rule has the general form **on** *event* **if** *condition* **do** *actions* and its declarative semantics specify that an action (or set of actions) must be executed provided an event occurs and a set of conditions hold. For example, using ECA rules we could specify that after a book has been inserted in a database, the author's information must be inserted in the corresponding `author` ta-

[1]ECA rules are also called *active* or *reactive* rules

ble. An alternative form of ECA rules allows the definition of a post-condition, which must be satisfied after the actions are executed. This leads to a type of rules called *Event-Condition-Action-Postcondition* rules (or simply ECAP). However, the same behaviour can be obtained if the post-conditions are considered inside the action part. Another type of active rules are the so-called *production rules*. These rules have the form of *Condition-Action rules* and they specify that, when the condition becomes true the actions are executed. In principle, production rules could be simulated by ECA rules where the event part is always the constant *true*. However, this approach is not valid due to the difference between their semantics. ECA rules execute *every time* the specified event occurs; provided stated conditions hold. Instead, production rules of the form condition-action execute *only once*, when the condition becomes *true*. If we assume that the actions in the ECA rules are idempotent then ECA rules are equivalent to condition-actions rules.

An alternative approach for implementing reactive behaviour is to use conventional programming languages. In this case, application programs encapsulate the reactive behaviour. Nevertheless, the use of ECA rules for expressing reactivity has some important advantages compared with the second approach, as mentioned in [12]. Firstly, ECA declarative rules provide a clear separation of concepts. Events, conditions and actions are easily identified and thus, evaluation and optimization of rules is easier to perform. Secondly, the use of rule bases containing a set of related ECA rules facilitates rules maintainability. Thirdly, ECA rules constitute an abstract paradigm suitable for modelling different types of reactive applications.

Triggers, a restricted form of ECA rules, were used in the implementation of reactive functionality in Active Databases. They operate at the database level by monitoring changes in the database's state and executing operations on it. Events specified in triggers reflect changes in the database and coincide with the update operations supported by the DBMS. Their actions are executed inside the database and correspond to the update operations of the database. Moreover, they are usually expressed using the programming language supported by the database. Compared to ECA rules operating at the application-level, triggers are low-level ECA rules. Nowadays, all the major DBMS vendors provide support for triggers.

2.1.2 Reactivity in the (Semantic) Web

As in the case of Active Databases, reactive behaviour in the (Semantic) Web can be implemented through the ECA paradigm. Besides the advantages already mentioned, ECA rules provide several additional benefits in this context. First, most of the real world situations involve the occurrence of events thus, a mechanism that considers events explicitly is needed in order to effectively model these situations. Second, active rules in general are not only easily understood by humans but also by machines. Third, events constitute a high-level communication paradigm which allows the exchange of

data between systems. This event-based communication requires appropriate mechanisms for dealing with events. Thus, ECA rules are more appropriate than production rules of the form condition-action. Fourth, the condition and action parts of rules may refer to remote data. Therefore, events can be used for transmitting data from one system (Web site) to another. Finally, ECA rules have been extensively studied in the field of Active Databases and hence, experience and results obtained in that area can be adopted and adapted when necessary.

In addition to the implementation issues considered when implementing reactive behaviour in active databases, the heterogeneous and distributed nature of the (Semantic) Web rises new important issues that must be considered:

- First, in the (Semantic) Web, atomic events may be update operations executed in some data repository or messages exchanged between two Web sites. In the same way, composite events may be more than simple combinations of update operations on data repositories. Composite events may reflect situations that occur globally in the Web, such as *"the cancellation of an order at an online marketplace followed by cancellation of delivery order at another site"*. That is, beside composite events that occur at the data repository level (traditional databases and XML or RDF repositories), we may have composite events that reflect situations occurring in the Web; i.e. they may combine different events that occur at different locations. In general, composite events may be high-level application-dependent and -independent events. Therefore, the ontology of events calls for appropriate event detection techniques and event algebras for expressing them.

- Second, actions in the (Semantic) Web can be composition of simple actions, i.e. composite actions. In this case, the constituent actions may be executed at different locations of the Web and thus, the composite action is said to be executed globally in the Web. Simple actions can be notifications, in the form of messages, sent from one application to another. Also, an action can be an update or query request sent from one application to another, possibly located at different locations. For example, as the result of placing an order at an online marketplace, the virtual store's Web site may send a notification to the customer, generate a shipment order and request a bank transfer. In this case, these three actions can be grouped to form a composite action called, e.g. *processSaleOrder*.

- Another important aspect is the specification of queries. In active databases, queries are commonly evaluated inside a single node or system. In the Web, queries may need to be evaluated in different nodes as they refer to portions of data that are stored at different locations. Moreover, queries can be evaluated against persistent data (databases) or against volatile data, i.e. event data. The last method allows for extracting information carried in events.

In the Web, a common approach for specifying reactive behaviour is to use rule-based reactive languages (also called active languages). Such languages use reactive rules for describing the reactive behaviour of a system and in particular, *Event-Condition-Action (ECA) rules*. Several proposals for describing reactive behaviour using active languages exist in the literature, such as Active XQuery [10], RDFTL [58], the General Language for Evolution and Reactivity in the Semantic Web [53], an ECA Language for XML [12] and XChange [63] among others.

2.2 General Aspects of Active Rules

In this section we briefly discuss some of the important aspects related with the use of active rules for implementing reactive behaviour.

Languages for Active Rules

In order to effectively use Active rules, a language providing a syntax for specifying such rules is needed. In the same way that Prolog provides a syntax for defining deductive rules in terms of their head and body, a language for active rules must provide the primitives to define rules in terms of their components. For example, we could conceive an active language that uses primitives such **ON**, **WHEN(IF)** and **DO(THEN)** to define an active rule of this form:

ON *event* **WHEN** *condition* **DO** *actions*

Furthermore, a language for expressing each of the components of a rule is needed. That is, we need an event language providing the means for defining event expressions, a condition language for defining boolean conditions (using e.g. boolean connectives) and an action language whose expressions denote actions. In this respect, most of the cases fix the language used for defining these components, i.e. the active language at hand embeds a (sub)language for defining components of rules. Every rule defined in the active language uses the same (sub)language for all its parts. An alternative approach [7] is to allow the use of different languages for specifying the components of rules. With this approach, a rule in a rule base may define an event using one event language, while another rule may define an event using a complete different event language. We will describe this approach in more details later in this work.

Analysis of rules behaviour

An important aspect to be considered when dealing with ECA rules is the analysis and optimization of a set of rules (*rule set*). A rule may trigger several other rules as its actions may rise new events, which in turn activate other rules. These triggered rules

may in turn trigger new rules and thus, there is the possibility for an infinite execution of rules. This ultimately affects the termination property of a set of rules. Moreover, a rule may trigger itself indefinitely if the execution of its actions keeps the condition true and rises the event associated with it. In the same way, conflicts among rules may exist if an event triggers more than one rule at the same time. In general, the behaviour of a set of rules is influenced by the type of events and actions that can be expressed with the rules. The problems of rule termination and optimization of rules have been extensively studied in the area of Active Databases [6, 13, 14, 16–18, 48].

Communication between components

During the processing of ECA rules two types of communications are involved from a rule's point of view. First of all, there is a vertical communication that happens between the ECA engine and the services or modules implementing the sublanguages associated with the rules' components. For example, the result of evaluating an event expression is an event instance together with event data (event parameters). The event parameters must be passed to the rule engine in order to continue with the execution of the rule. This may be implemented following different approaches, like e.g. system variables or variable bindings [7]. Second of all, there is horizontal communication that happens between the components of a rule. This happens for example when the results of an event evaluation are needed in the query part of a rule. Also, when a query expression is evaluated, the result of this evaluation may be required in the action part. For example, if the action is to send an email to a person, the email address may come from the database and thus, retrieved by evaluating a query on the database. The approach used in [7, 38] uses logical variables (variable bindings) as transition variables for this purpose, something that is similar to the approach used in Logic Programming and deductive rules. In summary, the implementation of rule engines, event detectors, query evaluators and action engines must consider these two forms of communication and provide efficient mechanisms for doing it.

Expressiveness of sublanguages

The expressiveness of the (sub)languages used for specifying the components of a rule is very important. It determines for example, the type of events the rule can react to, the type of queries that can be formulated and the type of actions that can be executed.

In the case of events, they can be classified into different categories such as low-level events, high-level application-dependent events, application-independent events and more generally, atomic and composite events. In order for applications to be able to detect and react to such a variety of events, it is necessary to have event languages which provide expressive sets of primitives for their specification.

In the same way, we may have an ontology of actions that considers low-level actions, high-level actions, application-dependent and -independent actions, among oth-

ers. Therefore, there is a need for expressive action languages that provide the appropriate set of primitives for dealing with different types of actions.

As for queries, an application may need to evaluate atomic queries (similar to atomic events) or composite queries. Queries can be evaluated on data repositories such as XML or RDF. In this case, the query language supported by the data repository can be directly used by the application. However, in the case of composite queries there is the need for algebras which allow the specification of queries in terms of other queries. Here, once again, the type of composite queries that can be expressed depends on the set of primitives provided by the language.

2.3 Related Work

2.3.1 Research on Active Rules

In the Web, ECA rules are being used in different contexts. In XML for example, several proposals for implementing reactive functionality on XML repositories exist. In [22], the authors argue that active rules represent a natural paradigm for the implementation of reactive services on XML repositories and propose the use of ECA rules for the rapid development of e-services. They also investigate the use of active rules in the context of XSLT [34] and the query language Lorel [67]. More specifically, they extend XSLT *template* expressions with event and condition parts in order to program reactive behaviour. Related to this work is [23], which proposes the use of ECA rules for detecting changes on XML repositories and reacting accordingly to them by sending information to interested remote users. Along the same lines, the work presented in [12] defines a language for specifying ECA rules on XML repositories. The monitored events reflect modifications to XML documents stored in the repository and the actions are update operations executed on the repository.

In the context of RDF, the ECA paradigm has been used for example in [58], where the authors define an ECA language for implementing reactive functionality on RDF repositories. In this case, events reflect modifications to the graph/triple representation of RDF documents and actions represent updates operations executed on the RDF repository. As we will see later in this work, all the active languages studied in chapter 4 are based on the ECA paradigm.

The analysis of ECA rules has been first studied in the field of Active Databases and a considerable amount of work exists [6, 13, 14, 16–18, 48]. More recently, the topic has been investigated in the context of XML. For example, in [12, 15] the authors discuss *triggering* and *activation* relationships for ECA rules in the context of XML. More specifically, they use the concepts of independence of XPath expressions and XPath containment to determine *triggering* and *activation* relationships among rules. Triggering and activation relationships are important because they can be used for analyzing

properties such as termination and confluence of a set of rules.

2.3.2 Update and Query Languages

In the context of the Web, active rules have been used for implementing reactive behaviour on top of XML and RDF repositories (see e.g. RDFTL [57,60], Active XQuery or the approach used in [22,23]). One of the characteristics of these approaches is the use of query and update languages for specifying the condition and action part of rules respectively. For example, actions can be update operations provided by the update language supported by the repository at hand. Following the same approach, the active language XChange [63] embeds the query language Xcerpt [3,66], which is used for accessing persistent data in the Web.

In some cases however, the underlying languages are not expressive enough as to be able to model a particular situation. For example, the update language may not include a move operation for moving complete subtrees in an XML document. In these situations, a possibility is to extend an existing query or update language in order to obtain a language that is more suitable for our needs.

Update languages are also used when specifying events in the event part of rules. In this case, the event language matches the update language supported by the repository and the event expressions are specified in terms of the update operations provided by the language. For example, an insert event denoting the insertion of data in XML or RDF documents corresponds to the insert operation found in the update language being used. This approach is similar to the one used for triggers in active databases.

Although most of the research regarding manipulation of XML documents has been focused on the development of query languages, several update languages for XML have also been proposed. For example, *XPathLog* [73], an extension of the XPath language [74], is a declarative, rule-based language for querying and manipulating XML data. Another example is the *XML-RL Update language* [51] that extends XML-RL, a declarative rule-based query language for XML, in order to provide Update capabilities over XML documents. Other approaches to updating XML documents include [70, 77]. The work on query languages for XML includes *XML-GL* [26] (a graphical query language for XML), *XQuery* [20], XML-QL [37], XMLQuilt [29] and Xcerpt [3,66], among the most important ones.

The adoption of RDF as the standard language for representing information in the Web calls for the implementation of query languages capable of extracting data from RDF documents. Research on query and update languages for RDF includes RQL [49], RDQL [75] and SPARQL [76], among the most important ones.

Chapter 3

Event Detection: General Aspects and Techniques

Contents

3.1 Event Model **16**

3.2 General Aspects of Event Detection **18**

3.3 Techniques for Event Detection **21**

3.4 Related Works **31**

In this chapter we turn our attention to the detection of events. More specifically, we define the concept of Event and classify them in an ontology according to their types. After this, we report on the general aspects of event detection, exploring and discussing different techniques used in the area of Active Databases and the Web. Finally, we present some previous works on this topic in the area of Active Databases as well as in the Web.

3.1 Event Model

An event is something that happens at a given point in time and at a given place, e.g. a DBMS or a Web site. In other words, an event is a happening that is considered to be relevant for some application. Events can be classified according to different abstraction levels. For example, they can be low-level events that usually occur at the database level. These types of events reflect changes in the state of a database (relational or object-oriented databases and XML/RDF repositories as well). These events, sometimes called *database events*, are usually raised by database operations like for example insert or update operations of the SQL language. Common event types in this scenario are method events (in Object-Oriented Databases) or transaction events, which describe transaction behaviour. At another level of abstraction we have higher-level, application dependent events such as *the cancellation of a flight reservation at an airline company's Web site*. Furthermore, events can be application-independent situations like e.g. *the reception of a message* from another Web site or system. The events just described are usually known as *atomic events* or *primitive events* [1].

Besides atomic events, we can also identify combinations (temporal-based or structural-based) of other events. For example, the situation where the cancellation of a flight is *followed by* the cancellation of a hotel reservation, which may occur at a travel agency's Web site. These type of events are usually referred to as *composite events*. That is, a composite event is an event that is defined in terms of other (sub)events; every (sub)event may be an atomic or composite event.

As we have seen, events can be classified into different classes and an ontology of events can be defined. The definition of such ontology is important because it helps to identify the classes or types of events that can be detected by an event detector and their relationships. Figure 3.1 on the facing page depicts the ontology of atomic events presented in [7]. Here, we extend it with the definition of composite events, leading to an ontology for events in general.

Atomic events can be classified into *application domain events* and *application-independent events*. The first class of events are defined by an application domain ontology. The second class can be further classified into *data-level atomic events* and *application-independent domain events*. Application-independent events are defined by an application-independent domain ontology. Furthermore, we can identify different types of atomic events. For example, *explicit events* and *temporal events* are also considered atomic. Explicit events, or sometimes called remote events are detected outside the system (by other detectors) and signaled to the system by messages or any other means of communication. Note that explicit events can be composite events for one event detector but atomic for others. Temporal events are related to time and require the use of a system's clock in order to be detected. Figure 3.2 on the next page shows

[1] the terms primitive and atomic events are usually used to refer to the same type of events

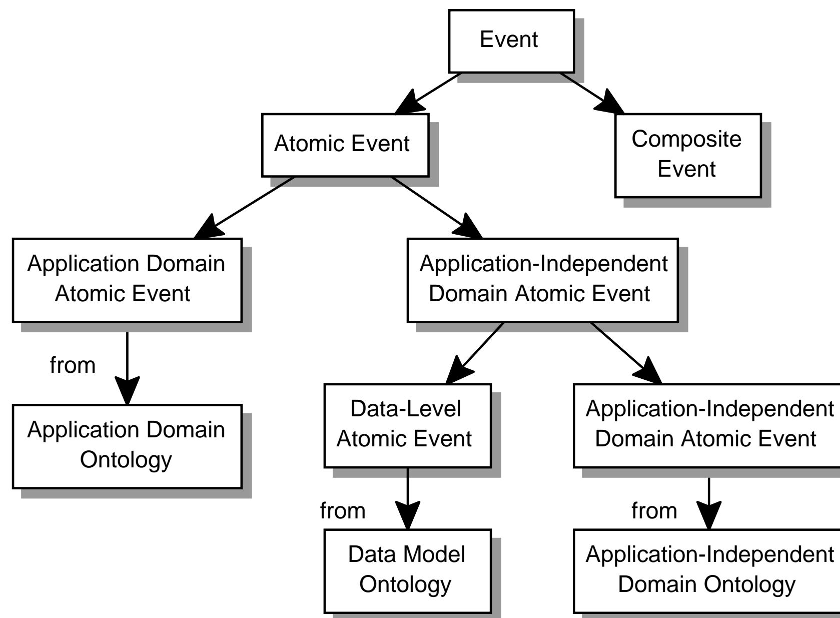

Figure 3.1: Ontology of atomic and composite events.

a classification of atomic events. Note that every reactive application is free to define any additional atomic event types that may need.

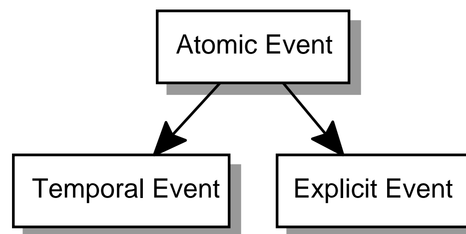

Figure 3.2: A possible classification of atomic events.

Application domain events. Also called application-dependent events, they are events that belong to the domain of a reactive application. For example, the cancellation of a flight in the domain of travel agencies is an application domain event. It can be specified using the expression *flightCancelled(flightNr,date,from,to)*. These events are high-level, application-dependent events and they are defined in an application's ontology.

Data-level events. They represent changes in a database and are raised by database operations. For example, in an object-oriented database, the execution of a method raises an event (usually referred to as *method events*). In relational databases, insert operations raise insert events. Data-level events are defined in the data model ontology.

Application-independent domain events. These events are not defined in the application domain ontology but in other ontologies, like for example the messaging ontology or the network communication ontology, etc.

3.2 General Aspects of Event Detection

In the context of Active Databases and the (Semantic) Web, an event detector is a software system capable of detecting events that occur in a given environment. The simplest scenario is a single system or database where all the events are local, i.e. they occur and can be detected inside the system. A more complex scenario is a distributed environment, where events may occur at different places and thus, need to be communicated among event detectors. This is the case of the (Semantic) Web where a set of autonomous information systems interact with each other by exchanging information about different domains. In this case, an event raised at one Web site may be relevant at another Web site.

Basically, an event detector works as follows: It accepts event expressions from subscribers describing the events to be detected. It stores the event definition and then, when an event occurs and is detected, the event detector reports the occurrence to the appropriate subscribers. The basic functionalities that should be provided by an event detector are the folowing.

- *Registration of event expressions.* An event detector must provide a mechanism for event registration, i.e subscribers (e.g. ECA engines or Active DBMS) must be able to send event expressions to the event detector so as to specify the relevant events they are interested in.
- *Communication of event parameters.* An event detector must provide a mechanism for the computation and exchange of event parameters. An interface with subscribers must also provide means for exchange of event-related information. For example, for receiving predefined values for some of the parameters.
- *Notification of to subscribers.* After an event detector detects an event, subscribers must be notified of its occurrence. A notification mechanism based on messages for example should be implemented.
- *Event detection.* The detection of events itself can be implemented using different techniques. Here, the type of events being monitored and the properties of the environment (distributed versus non-distributed) play an important role. In the case of composite event detection, a mechanism for detecting or receiving notifications about atomic events must be in place.

Although it may seem simple, there are many aspects that should be considered when implementing an event detector. Among the most important ones we identify the following:

- *Specification of events.* Primitive or atomic events are usually specified by a name and a list of parameters. For example, an event reflecting the availability of a new book in an online book store can be specified by the expression

`newBook(BTitle,BAuthor,BPrice)`. Here, *newBook* is the name of the primitive event and *BTitle,BAuthor,BPrice* are the formal parameters. As for the case of composite events, they are usually defined by expressions of an event algebra, which provides composers for combining component events. For example, in SNOOP, the event expression `newBook(BTitle) ; newCD(CArtist)` defines a composite event. Here, *;* represents the sequence operator, while *newBook(BTitle)* and *newCD(CArtist)* are the constituent events.

- *Event algebras.* In the case of composite events two alternatives are possible. On one hand, we could use model-independent event algebras such as SNOOP. These event algebras can be used for modelling events in different target domains. On the other hand, several implementations of event detectors define their own event language, tailored to specific needs; mostly due to the types of events the event detectors deal with. These event languages are usually based on previous ones, e.g SNOOP, and extend the base language by adding new operators to model different situations, thus extending its expressivity. As an example, the active language XChange [63] defines its own event algebra for specifying events.

- *Event detection at different levels.* In the (Semantic) Web, we can identify two different levels where event detection takes place. At the *lower-level* or *database level*, event detectors are concerned with the detection of events that reflect changes in XML/RDF repositories or relational and object-oriented databases. Then, at a *higher-level*, event detectors are concerned with the detection of application-dependent events, like e.g. *"the cancellation of a flight reservation"*. Note that high-level, application-dependent events may be implemented as a combination of lower-level events and thus, a translation or derivation mechanism for events may be needed. For example, a high-level event such as *updateFlightTime(FlighNr,From,To,DepartureTime)* may be implemented as a delete operation followed by an insert operation in the database (although unlikely as the update operation serve this purpose). Derivation of events also occurs between events of the same level.

- *Detecting primitive and composite events.* In general, event detection requires two types of event detectors. First, we need to implement primitive event detectors capable of detecting temporal events and data-model events. Usually, primitive event detectors make use of the facilities provided by the underlying system so as to be able to detect temporal events. For example, using the operating system's primitives. Second, composite event detectors implement the semantics of the event language's operators and detect events by processing and combining the incoming primitive events.

- *Communicating event parameters.* One of the most important aspects in event de-

tection, specially in distributed environments, is the ability to represent and communicate event-specific information. As we mentioned before, representation of event-specific information is achieved by means of event parameters. Now, when events are communicated between systems, the parameters values must be also communicated. Furthermore, in the context of ECA rules, the condition and action components may need to access the event-specific information. Thus, we need to define a mechanism for exchanging the event-specific information, i.e the event parameters. In general, event detectors are free to implement this using different techniques, however, as we will see later in this work, the choice of a particular technique influence the integrability of the event detector.

- *Event Detection Semantics.* While *primitive/atomic* events occur at a given point in time (atomically), composite events occur over an interval; i.e. they have both a starting and ending point. In other words, a constituent event initiates the composite event and then, another constituent event finalizes it. As a result of this, primitive events are detected at the same time they occur, i.e. the occurrence time of a primitive event coincides with its detection time. On the other hand, composite events can be detected according to two different semantics [4]. The *detection-based semantics* detects composite events at the end of the interval over which they occur. In other words, the detection time of a composite event corresponds to the detection time of the last constituent event that has been detected. Under this semantics the event's occurrence and detection times are considered the same. Conversely, the *interval-based semantics* detects composite events over an interval. It considers the starting and ending point of the interval over which a composite event occurs, whereas the *detection-based semantics* considers only the ending time. With *interval-based semantics* an event's occurrence time differs from the event's detection time.

  Most event specification languages implement the detection-based semantics. However, as mentioned in [4], the use of this semantics leads to incorrect detection of composite events in some cases. In general, the problem with detection-based semantics is that it does not capture the correct semantics of some event expressions; under certain combinations of events. This is due to the fact that composite events are detected by considering only the ending time of their intervals and discarding the starting time. Consider the following example.

Example 3.1 ***(Detection-based semantics and Interval-based semantics)*** *Consider the event expression in SNOOP* `E1 ; (E2 AND E3)` *and suppose that atomic event instances* `e1`*,* `e2` *and* `e3` *occur at time* `t5`*,* `t3` *and* `t6` *respectively. Under detection-based semantics, composite event* `E2 AND E3` *is detected at time* `t6`*. Then, the composite event* `E1 ; (E2 AND E3)` *occurs at time* `t6` *as atomic event* `E1` *occurs before* `E2 AND E3`*; i.e the condition for* `;` *is satisfied (*`t5` *is less than* `t6`*). However, if*

we consider the time interval over which the composite event `E2 AND E3` *occurs (*`t3` *to* `t6`*), the event* `E1` *does not occur before* `t3` *and thus, the detection is incorrect. On the other hand, if we considered the interval-based semantics, the composite event* `E1 ; (E2 AND E3)` *would not be detected as the event* `E1` *does not occur before* `E2 AND E3`*; i.e.* `t5` *is not less than* `t3`*. Thus, in this case the composite event is not detected and the result is as we expected.*

3.3 Techniques for Event Detection

The techniques used for detecting events depend on the type of events one is interested in detecting and on the domain where events are being detected (e.g. Databases, XML/RDF repositories or the Web). In general, we can classify the strategies for detecting events into those used for detecting composite events and those for detecting primitive/atomic events.

A strategy for composite event detection implements an event algebra's semantics. More specifically, it provides a set of data structures used for storing event data and a detection model. A detection model specifies how simpler events (primitive or composite) are combined in order to form composite events. It considers the semantics of the event algebra's operators as well as the restrictions (e.g. temporal restrictions) imposed to the set of constituent events. Therefore, strategies for detecting composite events are mainly influenced by the types of operators provided by the event algebra and by the semantics of event detection implemented, e.g. *detection-based* or *interval-based* semantics [4].

Conversely, a strategy for detecting primitive/atomic events does not implement an event algebra and depends on the type of primitive events to be detected.

3.3.1 Detection of Primitive Events

The technique used for detecting primitive/atomic events depends on the type of primitive events being considered. For example, in the context of Active Object-Oriented Databases, a *method event* is a primitive event that describes invocations of methods in objects; these events may refer to class methods or instance methods. A common approach used for detecting this type of events is to implement a *wrapper* that captures the method invocation signal and executes an action that raises the corresponding event. For example, if the method `update_salary(newSalary)` has a method event *salary_updated* associated with it, a wrapper would capture the invocation of the method and raise the primitive event *salary_updated*. One possibility here is to modify the method's code in order to introduce the raising sentence inside. However, this would require to recompile the method's class every time an event is associated with the method. An alternative solution is to extract the raising sentence and put it in the

wrapper outside the method's code. This approach to primitive event detection was used for example in the Active Object-Oriented Database System SAMOS [40].

Transaction events, which reflect execution of transactions in Active Databases, can be detected by implementing a *wrapper* that captures transaction methods such as `init_transaction`, `end_transaction` or `commit_trsnsaction`. This mechanism is suitable for cases where transactions are implemented as classes (usually in OO Databases). As for *temporal events*, an event detector needs to use the system's clock functions.

3.3.2 Detection of Primitive Events in XML

Event detection using DOM Event Model

A first alternative to detect events on XML repositories is to use the DOM (level 2) Event Model [36]. It defines an API that provides programs with an event system so as to detect events that occur in a document. Among the different types of events defined by the DOM Event Model we find the so-called *mutation events*, which are insertion and deletion of elements as well as insertion, deletion and update of attribute values and PCDATA content. Mutation events occur when the structure of a document is modified. Using the DOM Event Model it is possible to associate event listeners with the nodes of a document and detect events by monitoring methods calls. This approach has been used in different implementations, like e.g. in [19] where the DOM Event Model is used for supporting the detection of composite events on XML documents (DOM events are regarded as atomic events). An important disadvantage of this alternative is that events are associated with document instances rather than with document schemas. That is, high level, schema-dependent events like the ones proposed in [19] must be translated into instance-dependent events.

Event Detection using Repository Operations

When the XML repository provides operations for data manipulation such as insert, delete or update, external applications can use these operations in order to modify the documents stored in the repository. In this case, event detection is performed inside the repository by monitoring operations calls. That is, an event detector is placed on top of an existing XML repository and every data manipulation issued to the repository is catched by the event detector. This approach allows the detection of *explicit events*. Examples of XML repositories providing this type of operations are [1,2].

Alternatively, external applications may define their own set of events or external events. In this case, if external events are not directly supported by the operations provided by the XML repository, then they must be translated into a sequence of operations in the repository. For example, an external application may define a *move event* that occurs when a subtree is moved from one position to another inside the document.

In this case, if the repository does not provide a *move* operation reflecting this external event, the event must be translated into a sequence of *delete* and *insert* operations. This mechanism allows external events to be treated as explicit events.

The main disadvantage of this approach is the tight integration between the event detector and the repository itself. This in turn affects the modularity and extensibility of the event detector. However, the advantage os this approach is that events are detected as soon as they occur; i.e. when the data manipulation operation occurs.

XML-Diff Algorithms

In the context of the (Semantic) Web, manipulation of XML documents is sometimes performed by applications outside the XML repository, i.e without using the repository's data manipulation operations. For example, an application may retrieve an XML document, modify it by means of application-defined operations and then send it back to the repository for storage. In this case, XML repositories are unable to detect changes in the stored documents as modifications to XML documents are performed by external applications and hence, no repository operations are executed.

A solution to this problem is to detect changes by comparing two versions of the same document. That is, one could implement or use an algorithm, usually called *XML-Diff*, that compares the old and new versions of a document and computes the sequence of transformations or operations that must be applied to the old version so as to obtain the new version. Detection of changes in semi-structured data has previously been studied in [30–32, 35] and several alternatives exist.

One possibility is to implement an *XML-Diff* algorithm that produces as output an optional, one-to-one identity relationship between nodes of different versions. An essential prerequisite for this is to have a mechanism for identifying nodes in XML trees, e.g using XML ID attributes or their position in the tree. After this, events can be detected (derived) by analyzing the relationship between nodes. For example, if a node in the old version is not related with any node in the new version, then this means that the node has been deleted from the document. In the same way, if a node in the new version is not related to any node in the old version, this means that the node has been inserted and thus, an insert event is detected. Let us illustrate this with an example:

Example 3.2 ***XML-Diff with identity relationship.*** *Consider the two versions of an XML document (represented as a data tree) shown in Figure 3.3 on the following page. The result of running the XML-Diff algorithm on them is the identity relationship depicted as arcs between nodes in the figure. From this, an event detector can derive the deletion of nodes identified 4 and 5.*

The previous alternative is applicable when the fragments of data being compared can be uniquely identified. But, when no means for unique identification of data are

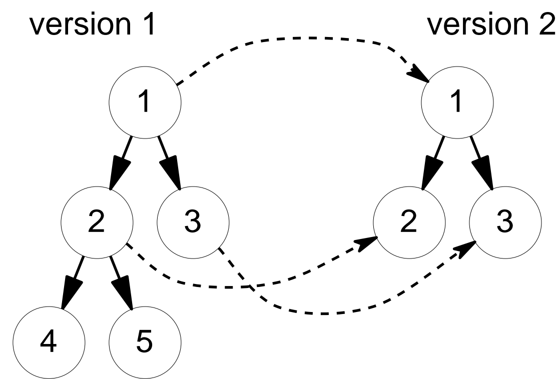

Figure 3.3: Two versions of an XML document. The older version is on the left-hand side

provided we need another approach. The work presented in [30] does not rely on the usage of identifiers to uniquely identify fragments of data, although it can benefit from them if they are provided. It implements an *XML-Diff* algorithm that generates an *edit-script*. An *edit-script* contains the sequence of update operations that transform the old version of a document into the new version of it. That is, entries in the edit-script represent insertions or deletions of XML elements and insertions, deletions or updates of attributes and PCDATA values. Note that there may be several equivalent edit-scripts for two versions of the same document. Once the edit-script has been generated, an event detector can use the information on it to try to detect (derive) the events that have occurred. Events detected using edit-scripts are usually called *edit-script events*.

One important aspect that must be considered when detecting events using the *XML-Diff* approach is the order of data. Compared to the object-oriented and relational models where order of data is not important, the order of elements in XML is relevant (e.g. an XML schema may restrict the structure of elements), while order of attributes is not. An *XML-Diff* algorithm implementing an *order-independent semantics* was proposed in [33]. The algorithm generates a *minimum-cost edit-script* which, besides traditional data manipulation operations, includes *move* and *copy* operations; thus providing more meaningful results. Moreover, rather than assuming the existence of node identifiers to match nodes of both trees, the algorithm compares the content of nodes in order to match them (although the algorithm can benefit from the existence of node identifiers). A much simpler version of this algorithm is presented in [30]. It implements an *order-dependent semantics* and thus the order of data is relevant. Also, *copy* operations are not considered by the algorithm.

Another alternative is the implementation of the *XML-Diff* algorithm presented in [35]. The distinctive feature of this implementation, not present in previous approaches is the use of *deltas* to represent changes. A *delta* is an XML document representing changes between two versions of the same document. Furthermore, the algorithm supports an additional move operation on subtrees, in addition to the classic operations supported by other approaches. The algorithm works by detecting and

matching subtrees that remained unchanged between two versions of the same document. It then considers ancestors and descendants of matched nodes in order to match more nodes. The matching of nodes relies on the use of XML ID attributes to identify nodes.

3.3.3 Techniques for Composite Event Detection

Two aspects are important when detecting composite events. The first one is related with the representation of event expressions. For this several techniques exist which model event expressions with different data structures. Here, the choice of one of them is influenced by the efficiency and the type of information to be stored in the data structures. The second aspect is how to process and combine incoming atomic events in order to evaluate event expressions, i.e detect composite events. In this case we have two alternatives:

- *Non-incremental evaluation* This technique evaluates the composite expressions every time a new atomic event is signaled to or detected by the system. That is, upon atomic event reception the event detector checks which of *all* the composite expressions use the current atomic event. When another atomic event arrives, the detector checks every composite expression *again*. However, in every case the event detector must check if *all* the constituent atomic events of a composite event have occurred. Moreover, the temporal conditions (or any other condition) among constituent events must also be checked. This technique requires the event detector to process the same atomic event more than once.

- *Incremental evaluation* The alternative to the previous technique is to perform an incremental evaluation of a composite expression. With this approach, the sequence of constituent events is processed incrementally in a step-by-step fashion. This is analogous to the bottom-up techniques used by compilers with the difference that, instead of a string of symbols the event detector processes a sequence of atomic events. When an atomic event is detected, the event detector checks which composite expressions the atomic event contributes. If another atomic event arrives, the event detector process only the current atomic event, i.e ignoring the previous ones as they were already processed. The advantage of this approach is that atomic events are processed only once. Clearly, incremental evaluation is preferred to non-incremental evaluation.

Several techniques exist for implementing incremental detection of composite events. Note that several implementations exist that use the same technique but, although the efficiency and the additional data structures may differ, the general idea of the technique is always the same.

Using Event Trees

Several event detectors, like [19,28,38,52,56,78], model event expressions using *event trees*. In the tree-based approach, event expressions of an event algebra are represented as trees. Leaf nodes denote atomic or primitive event types, internal nodes represent the language's operators and the tree's root is the outermost operator of the expression. In general, leaf nodes represent the type of events (event types) that can not be detected by the system (composite event detector). That is, events are detected outside the system and then signaled to the composite event detector. The work presented in [56] follows this approach, where leaf nodes represent external events occurring outside the system and detected by other event detectors. When these events are signaled, the composite event detector uses them to detect composite events. Due to its algebraic nature, event expressions can be easily represented using trees. Furthermore, trees are a simple and efficient way of representing complex expressions and several algorithms exist and can be reused to work with trees.

One of the main advantages of using event trees to model event expressions is that we can combine or reuse event trees. Instead of representing every event expression with a separate event tree, an event detector may reuse other event trees and aggregate them in an event graph. In other words, if two event expressions *A* and *B* use the same subexpression *C*, the event tree representing the expression *C* is shared by the event trees representing the expressions *A* and *B*. Thus, an event graph is a collection of event trees, where leaf nodes can be shared by different internal nodes and operator nodes can also be shared by different nodes. Let us illustrate this with an example.

Example 3.3 ***(Event Trees and Event Graphs).*** *Consider the event expressions (in the SNOOP event algebra)* `A = newBook(BAuthot,BTitle) AND newCD(CTitle,CArtist)` *and* `B = newCD(CT,CA) ; newMovieDVD(DTitle,DGenre)`. *Figure 3.4 on the next page depicts the event graph representing both expressions. Expression A is represented by the event tree A, while event expression B is modelled by the event tree B. Note that the leaf node representing the primitive event* `newCD` *is reused by both event trees.*

The advantage of aggregating event trees is twofold. First, it allows for reducing the space required to store the information about an event expression. Second, it facilitates the detection of composite events as a primitive event instance can be used for detecting two different composite events.

Using event trees, event parameters can be stored at the leaf nodes an propagated to their corresponding parents when a primitive event is detected and signaled to the system.

The detection of composite events follows a bottom-up process that starts when an atomic event instance is signaled to the system. The event detector processes every incoming primitive event and stores its parameters in the appropriate leaf node. After

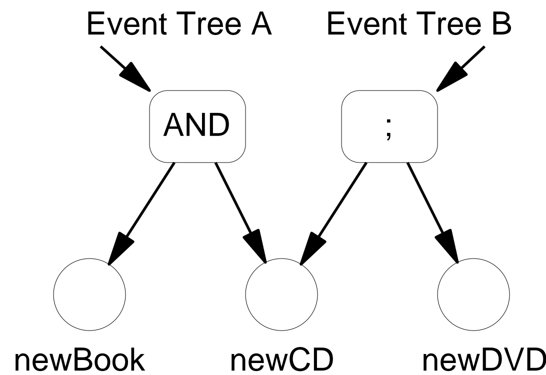

Figure 3.4: Combination of two event trees.

this, the detector activates the parent nodes and passes the parameters' values to them. At this point, the event detector executes the algorithm implementing the operator's semantics. Event instances are propagated from the leaves up to the event tree's root. When an event instance reaches the root of an event tree, the event detector signals the composite event together with the parameters. Note that additional computations and data structures can be performed and employed by a particular implementation. Moreover, the semantics of the language's operators influences the behaviour of an event detector. However, the general processing of an event detector using event trees is basically the same. For example, when an atomic event is signaled to the system and stored at the appropriate leaf nodes, an optional condition checking can be performed in order to validate the event instance.

Using Petri nets

Another technique used for implementing composite event detection is based on Petri nets. Petri nets are a well-known formalism used for modelling complex systems and specially the computations carried out by them. In the context of event detection, Petri nets are used for modelling complex events definitions and for implementing composite event detection in an incremental way. One of the advantages of using Petri nets for event detection is that they can easily model the flow of parameters values from constituent events to composite events.

As an example of the use of this formalism, we have the work presented in [41], which modifies Colored Petri nets obtaining the so-called S-PN or SAMOS Petri nets (used for implementing event detection in the SAMOS system [40]).

Without giving too many details [2], a S-PN comprises a set of places, which can be *input places* or *output places*. Both types of places model event expressions (also called event patterns or event definitions). Input places model constituent events of composite events, while output places model composite events. In addition to this, a S-PN also includes a set of auxiliary places used for modelling dependencies among events (e.g. time restrictions among constituent events). In order to represent event parameters,

[2]Additional information can be found in [68, 69]

S-PN use a data structure called *token*. A token stores the values of the parameters associated with the event pattern modelled by the place where the token is stored. Note that a place may contain several tokens. For example, when a primitive event occurs, the values of its parameters are saved in a token and the token is stored at the place representing the event definition of the event that has just occurred. Additionally, S-PN use *arc* and *guard* expressions. *Arc expressions* allow to compute the parameters of composite events based on the parameters of their constituent events. They contain variables, which are used for communicating tokens between places and transitions; i.e. variables store tokens information. Guard expressions instead, are used for expressing conditions or restrictions on constituent events.

Petri nets can also be combined to represent composite events. In other words, Petri nets representing constituent events can be combined (reused) into another Petri net that represents a composite event.

Example 3.4 ***Combining Petri nets.*** *Consider the event expression* ***E2 = E1 AND newDVD(DTitle,Director)****, where* ***E1 = newBook(BTitle,Author) AND newCD(Artist,Year)****. The Petri net depicted in Figure 3.5 models the composite event expression E2. It combines a Petri net for E1 and a Petri net for expression* ***newDVD(DTitle,Director)****. In the figure, circles represent places (colored circles are output places) while vertical bars represent transitions. Parameters for primitive events are stored in their respective circles.*

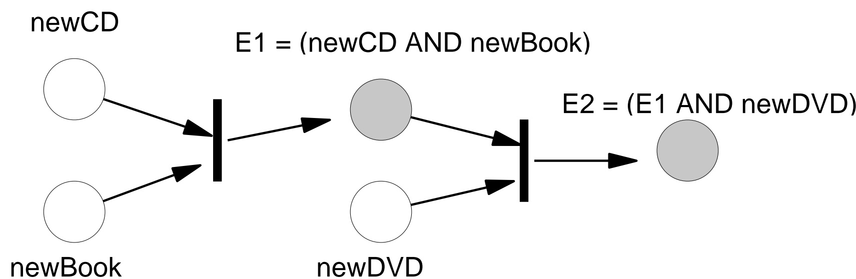

Figure 3.5: Combination of two Petri nets

Composite event detection using Petri nets is achieved by processing incoming primitive events. After a primitive event is signaled, the composite event detector marks the appropriate input place with a token. The token contains the event's parameters values. After that, the event detector checks which transition can be fired. Note that a transition fires when all its input places are marked, i.e contains tokens, and its guard condition is satisfied. Once a transition fires, all its output places are marked. At this moment, the composite event modelled by each output place is detected and the parameters' values are stored in the tokens.

The main advantage of using Petri nets for implementing composite event detection is that, as with the tree-based approach, they allow the implementation of optimization techniques for event expressions. Common (sub)expressions within a single

event expression or among different event expressions can be reused by combining single Petri nets (see example 3.4). Also, rewriting techniques can be implemented in order to transform complex event expressions into simpler equivalent ones. Rewriting techniques are specially interesting when event expressions are registered with an event detector as equivalent expressions need not to be registered twice. However, as pointed out in [47], Petri nets are more inefficient than other approaches.

Using Finite State Automata

An alternative technique for detecting composite events expressed by expressions of an event algebra is to use *Finite State Automata*. This formalism has been used for example in [44], where event expressions are implemented using finite state automata. The approach is based on the fact that event expressions are equivalent to regular expressions and hence, they can be executed or recognized by finite state automata. The approach uses *Deterministic* FSA as they are more efficient than the *non-deterministic* ones and can handle negated events in a better way.

In the same way that finite state automata recognize strings of symbols, a finite state automaton (FSA) detects composite events by recognizing the event expressions that represent them. That is, the input string for such automaton is the sequence of incoming primitive events. After a primitive event is signaled to the system, the FSA "reads" or processes the incoming primitive event and moves to the next state. Now, if the next state is an accepting state, then the event represented by the FSA is detected. The construction of a FSA is similar to that for regular expressions and it depends on the form of the event expression being implemented (or to be recognized); taking into account the operators used in it.

The automata-based strategy works fine when primitive events do not include parameters. However, when events include parameters and parameter computation is required, additional data structures must be considered. The solution implemented in [44] uses multiple automata in order to represent events that contain parameters. The disadvantage of this approach is that the resulting automata may contain a large number of states; thus reducing the efficiency of the event detection process. Additionally, by combining equivalent states it is possible to reduce the number of states required to represent an event expression; this in turn allows the optimization of event expressions.

3.3.4 Composite Event Detection in XML

Although composite event detection has long been considered in the context of Active Databases, its application to the XML context is not trivial and requires further investigation before existing approaches can be effectively adapted. The work presented in [19] considers three important aspects that characterize XML documents and pro-

poses an approach to detect composite events in XML that takes these aspects into account. More specifically, they identify the following features:

- *Temporal and hierarchical order of events.* Although in most of the cases events are ordered according to their time (occurrence time), an alternative approach is to order events according to their hierarchical structure. For example, if we consider an XML document where element `<b>` is a child of element `<a>`, the event produced by the insertion of element `<b>` is hierarchically related to the insertion of element `<a>`. Hierarchically related events are the most common type of events in XML.

- *Schema-based events.* A schema may restrict the number of elements inside another element (using *occurrence constraints* [39]). An important class of events that should be detected is the class of schema-based events. These events occur when an update operation satisfies these constraints, i.e when a constraint is satisfied. Approaches considered so far do not take these types of events into account.

- *Hierarchy of event types.* Event types can be hierarchically related and this can be exploited in order to allow for more expressive event type definitions. This also help to provide more reusable event types.

The approach presented here refines the event algebra SNOOP [28] by defining an abstract model for XML data, for XML events and for event types. Additionally, the underlying language is extended by defining a new parameter context, called *hierarchical context*, which allows the detection of composite events when their constituent events are hierarchically related, i.e ordered by their hierarchy rather than by their occurrence time. The language is also extended by the introduction of a new multiplicity operator, which allows the detection of *schema-based events*. Finally, a third extension is provided by the definition of new operators modifiers that extend the expressiveness of event type definitions.

Events reflect manipulation of nodes in an XML tree. Primitive events can be *insertions* and *deletions* of element, attribute and text nodes and *updates* of attribute and text nodes. A primitive event type is specified by the operation that causes the event and by the *path type expression* that identifies the node in the XML tree where the event occurs. Path type expressions identify nodes in the XML tree using type information, i.e at the schema level. They are based on path expressions. Composite events are defined by expressions of the language, using the language's operators and combining primitive and composite events.

Detection of primitive events is implemented using the DOM (level 2) Event Model. Instead, composite event detection is implemented using event trees, taking into account the operators' semantics and the hierarchical relationship between event types.

3.4 Related Works

In this section we take a look at the research activities done along four areas of interest: Event Detection in Active Databases, Event Languages, Event Detection in the contexts of XML and RDF and Event Notification Systems.

3.4.1 Event Detection in Active Databases

Event detection has been extensively studied in the field of *Active Databases* [61, 72], where the goal is to detect changes in the database's state. In this area, event detection was studied in the context of active rules for databases or, also called*triggers*. A considerable number of work has been done in the field of Object-Oriented Databases. For example, [44,45] considers the specification of events in active OODB and proposes the use of *Event-Action* rules, a variation of the classical *Event-Condition-Action* rules where the *condition part* is embedded in the event part. The authors show that the expressivity of expressions in the language is equivalent to that of regular expressions; when event expressions are evaluated over a stream of events. This leads to the use of *finite state automata* for implementing event detection. Also related is the work done in [42], which investigates composite event detection in the context of the system *COMPOSE*. Further research in this area was done in [40, 43, 71].

3.4.2 Research on Event Languages

Regarding event languages, SNOOP [28, 55] is one of the first attempts to define an event algebra for specifying events in active databases. The language provides a set of composers for combining atomic and composite events and thus, specifying situations that occur when; not one but a combination of events occur. Along the same lines, the work done in [68] investigates the specification and detection of events in the context of Object-Oriented Databases, more specifically in the OODBMS SAMOS [40].

As for distributed environments, the work presented in [52] defines a *Generalized Event Monitoring* language that allows for the specification of hight-level, abstract composite events. They are defined by combining simpler events that occur at different places; temporal constraints can be used for combining constituent events.

More recently, approaches to implement reactive functionality on the Web have been investigated. Most of the proposals define their own event algebra in order to model events or situations of a particular target domain. For example, the works done in [10, 19] investigate event detection in the context of XML. They define event languages for specifying events that reflect modifications on XML repository. Of particular importance is the approach taken in [19] where the authors consider the specification and detection of composite events on XML documents. Moreover, they argue that the definition of composite events whose constituent events are hierarchically related

has to be considered too. Furthermore, specification of events reflecting modification on RDF repositories has been addressed in [58][3].

3.4.3 Event Detection in XML and RDF

Recently, interest in event specification and detection has moved to the Web environment. In this context, event detection techniques are being investigated as they constitute the basis for implementing reactivity in the Web. Some of the approaches previously studied in the field of Active Databases are being reused in the context of XML and RDF repositories. However, due to the richer set of events that can be specified and detected in XML and RDF documents, those methods must be adapted. Change detection on XML documents and hierarchically structured data has been investigated in [30–33, 35]. More recently, event detection has been studied in the context of XML and RDF repositories. As we will see in this thesis, event detection in this context was considered in [10, 58].

3.4.4 Event Notification Systems

Besides Active Databases, several other event notification systems have been implemented. These types of systems are important as they support the development of event-based services, specially in distributed environments such as the Web. Along these lines, the EPS system [56] is a general-purpose *Event Processing System* that detects composite events expressed by event expressions. The READY system [46] is a notification service that provides asynchronous notification of simple and composite events. Another example is the *Situation Manager*, a component of the Amit system [5], is a run-time monitor that processes incoming events (obtained from event sources), detects situations (combination of events) and then reports them to the interested subscribers.

[3] some of these languages are being considered in this thesis

Chapter 4

Active Languages for the (Semantic) Web

Contents

4.1 RDFTL: A Trigger Language for RDF 34
4.2 Active XQuery Language 36
4.3 XChange Active Language 40
4.4 A General Language for Reactivity in the Semantic Web 47

This chapter presents a description of the active languages considered in this work. Each of the languages is described in terms of the event language used for specifying event expressions, the type of events that can be expressed with them and the mechanisms used to exchange information among the rules' components.

4.1 RDFTL: A Trigger Language for RDF

RDF Trigger Language [58] is an event-condition-action language designed to provide reactive capabilities to RDF repositories. Using RDFTL we can define ECA rules on RDF metadata so as to be able to detect changes in RDF documents and react accordingly to them by executing the appropriate actions. For example, an ECA rule defined in the language may detect insertions of triples in a RDF document and acts accordingly by updating a triple in the same or another RDF document in the RDF repository. Changes and events occur at a local RDF repository and no means for distributed event detection or action execution is provided. That is, events occur locally at the system where the rule is defined. However, we will see that an architecture for supporting RDFTL rules in P2P networks has been developed.

The distinctive feature of the language is that reactive rules operate over the graph representation of an RDF document. In other words, the event and actions parts of an RDFTL rule refer to the nodes and triples of an RDF graph that were modified by the execution of an insert/delete/update operation. Other approaches have considered rules that operate over the XML serialization of the RDF document.

4.1.1 Definition of the Language

Since ECA rules in RDFTL operate over the RDF graph representation of the RDF data model, we need a mechanism to specify the fragments of an RDF document (RDF graph) addressed by each of the rules' components. In RDFTL this is accomplished by embedding a path-based query (sub)language that operates over the RDF graph. Using this query language we can express the event and condition parts of an ECA rule. For example, a path expression in the path language may be used to define an event expression denoting the insertion or deletion of certain nodes in the RDF graph.

Path Language

The syntax of the path-based language is similar to the syntax of the XPath [74] language, although its expressivity is more reduced. Every path expression denotes a set of nodes and this set can be filtered out by a boolean expression defined over qualifiers and path expressions. The abstract syntax is shown in Figure 4.1.

E ::= resource"("*uri*")" (/P)?
P ::= P/P | P"["Q"]" | target(*arcname*) | source(*arcname*)
Q ::= Q and Q | Q or Q | *not* Q | P | P = string | P ≠ string

Figure 4.1: Path expressions grammar

Here, *resource(uri)* is the resource denoted by the URI given as parameter, *target(arcname)* denotes the set of object nodes in a RDF graph related by property *arcname*

to the subject nodes in the context and *source(arcname)* denotes the set of subject nodes in the graph related by property *arcname* to the object nodes in the context.

Rules in RDFTL

In RDFTL, ECA rules are defined by specifying their *event*, *condition* and *action* parts. Conforming to the semantics of active rules, the event part specifies the event that is being monitored in the system, the condition checks whether the system is in a particular state and the action part defines the set of actions to be executed if the event occurs and the condition holds. The general form of ECA rules in the language is defined by the following grammar, where *VariableName* is any valid name for a variable. The event, condition and action parts are defined below. For a complete definition of the language's grammar the reader is referred to [58].

```
R ::= "on" [VariableDefinition "in"] EventPart
      "if" [VariableDefinition "in"] ConditionPart
      "do" [VariableDefinition "in"] ActionPart
VariableDefinition ::= "let" VariableName ":=" PathExpression
                       [, VariableDefinition]
```

The condition and action parts of a rule may refer to a system variable called $delta. This variable is used as a communication mechanism between the components of a rule. The information about the inserted or deleted nodes as well as the inserted, deleted or updated arcs is passed from the event part to the other parts of a rule by using this variable. Depending on the type of event the content of the variable may differ. Moreover, depending on whether the variable is referenced in the condition or action parts, the rule execution model differs.

Events and Event Expressions In RDFTL, an event can be the insertion or deletion of resources (nodes in the RDF graph), the insertion or deletion of triples and the update of triples. Events are defined by event expressions with the following syntax and semantics.

- `(INSERT | DELETE)` *e* `[AS INSTANCE OF` *class*`] [USING NAMESPACE` *ns*`]`. In this expression, *e* is a path expression of the path language embedded in the ECA language and denotes the set of nodes that were inserted or deleted. Optionally, we can define the name of the RDF Schema class to which the inserted or deleted nodes belong. An event defined by this expression occurs when the set of new (for an insert expression) or deleted (for a delete expression) nodes is included in the set of nodes that result from the evaluation of the path expression *e* and the inserted or deleted nodes belong to the class *class*, if defined. Here, the variable $delta contains the set of inserted or deleted nodes.

- `(INSERT | DELETE)` *triple*. This expression defines an event that occur whenever a triple is inserted or deleted. Here, *triple* is an expression of the form *(subject,predicate,object)* and denotes the triple to be deleted or inserted. Any of these components can be replaced by the symbol "_". For example, the triple pattern *(subject,predicate,_)* can be used to denote insertions/deletions of triples with any value in its object part. In this case, the variable $delta contains the set of subject nodes of the inserted or deleted triples.

- `UPDATE` *updtriple*. This expression defines an event that occur when a triple on the RDF graph is updated. Updates in this case are considered changes in a triple object's value. Here, *updtriple* is an expression of the form *(subject,predicate, oldobject* $\longrightarrow$ *newobject)* and denotes the triple whose object value has been changed from *oldobject* to *newobject*. As in the previous case, the wildcard symbol "_" can be used in any of the parts. Variable $delta contains the set of subject nodes of the inserted or deleted triples.

Condition part A condition is a boolean expression built using boolean connectives and path expressions.

Action part Actions in RDFTL can insert or delete resources (subject or object nodes) and insert, delete or update triples. The action part of a rule contains a sequence of one or more actions.

4.2 Active XQuery Language

Active XQuery [10] is an active language for XML repositories whose main purpose is to provide XML repositories with reactive capabilities. It does this by extending the XQuery language [20] with *active rules* or *triggers* adapted from the trigger concept in SQL3. In other words, rules in the language emulate the trigger definition and so they provide the means for detecting changes over XML repositories. Triggers in Active XQuery specify the *events* to be detected, the *conditions* that must hold in order to execute each trigger and the set of *actions* to be executed upon event occurrence.

The language assumes the existence of an update model for XML which provides a basic set of update operations over XML repositories. In particular, the definition of triggers it is based on the model proposed by [70]. This approach extends the XQuery language with a set of primitive operations for modifying the structure and content of XML documents.

4.2.1 Syntax of the language

Triggers in Active XQuery can be decomposed into different components. Among the most important ones we have the *event*, *condition* and *action* components.

The Event component. It identifies the event being monitored by the trigger, i.e the happening that makes the trigger to be fired or triggered. The language regards events as the result of the execution of triggering operations, which correspond to the update primitives of the underlying update language. In order to define an event we must indicate the triggering operation associated with it and the fragment of XML affected by the execution of the operation. The affected XML fragment is described by one or more XPath expressions. Each of these expressions may refer to different documents and when evaluated over XML instances they produce a sequence of nodes. As it is with the SQL3 trigger concept, the language allows the definition of *before-triggers* and *after-triggers*. The difference between them is the triggering order of the trigger w.r.t the event associated with it. In the first case, the trigger's condition and action parts are considered before the event actually occur. In the other case, the event must occur first in order to evaluate the trigger's condition part and then execute the trigger's actions.

The syntax used to define events in the language is as follows.

```
(BEFORE|AFTER)
       (INSERT|DELETE|REPLACE|RENAME)+
       OF FragExpression (,FragExpression)*
```

The keywords *insert*, *delete*, *replace* and *rename* denote the triggering operations associated with the event and the trigger. Keywords *before* and *after* define before-triggers and after-triggers respectively. *FragExpression* is the XPath expression denoting the affected XML fragment.

The Condition component. The condition part is optional and defines the conditions that must hold in order to execute the trigger's actions upon event occurrence. Conditions are specified by *XQuery's Where clauses* and introduced by the keyword *when*. Below is the syntax used to define a trigger's condition.

```
WHEN XQueryWhereClause
```

The Action component. It defines the set of actions to be executed on the system when the event occurs and the conditions hold. Actions include updates operations of the underlying update language and possible external actions like sending a message to another system. It is worth mentioning that the execution of a trigger may cause another triggers to be triggered. The syntax for defining trigger's actions is as follows.

```
DO (XQueryUpdateOperation|ExternalOperation)
```

In addition to these components, an Active XQuery trigger have an optional *triggering granularity* associated with it. This property influences the trigger's execution and classifies triggers into *statement-level triggers* and *node-level triggers*. A statement-level trigger executes once for each set of nodes resulting from the evaluation of the above XPath expressions. On the other hand, node-level triggers execute once for each node in that set.

A trigger may also define and use variables in order to support the exchange of information among its components. These variables, called *transition variables*, can be referred to in the condition and action parts of a trigger. For example, the system-defined transition variables `OLD_NODE` and `NEW_NODE` denote the XML fragment affected by the execution of a triggering operation. The content of these variables depends on the type of trigger being executed. With node-level triggers, both variables denote the node that was inserted, deleted, renamed or replaced by an update operation. With statement-level triggers, these variables refer to the set of nodes affected by an update operation; i.e. each variable contains a set of nodes. In both cases, the `NEW_NODE` variable denotes the new state of the affected fragment and the `OLD_NODE` denotes the old state of the affected fragment. Additional transition variables can be defined by an *XQuery-Let-clause*.

Finally, an optional priority can be associated with every trigger. Priorities help to establish an execution order among a set of triggers, especially when several triggers are triggered by the same event.

The complete syntax of triggers in the language is shown in Figure 4.2. Here, line 1 defines the trigger's name. Line 2 defines the optional trigger's priority. Lines 3 to 5 introduce the event part. Line 6 expresses the trigger's granularity. Line 9 defines the transition variables, while in line 8 we specify the trigger's condition and in line 9 we define the trigger's action part.

```
1 CREATE TRIGGER NameOfTrigger
2 [WITH PRIORITY signedIntegerNumber]
3 (BEFORE|AFTER)
4       (INSERT|DELETE|REPLACE|RENAME)+
5       OF XPathExpression (,XPathExpression)*
6 [FOR EACH (NODE|STATEMENT)]
7 [XQueryLetClause]
8 [WHEN XQueryWhereClause]
9 DO (XQueryUpdateOperation|ExternalOperation)
```

Figure 4.2: Triggers' syntax in Active XQuery

Let us see an illustrative example in order to grasp the trigger concept just defined.

Example 4.1 ***(Active XQuery trigger).*** *Consider an XML document (movies.xml) containing information about movies. Let us suppose that we want to detect insertions of movies in*

the database and then, if the movie's year is 2002, insert the movie title into another XML document (listing.xml). The structure of the documents is defined below.

```
<!-- movies.xml -->
<movies>
 <movie id="1">
  <title>Ice Age 2</title>
  <year>2006</year>
 </movie>
 <movie id="3">
  <title>Mission Impossible 3</title>
  <year>2006</year>
 </movie>
 ...
</movies>
```

```
<!-- listing.xml -->
<listing>
 <title>Ice Age 2</title>
 <title>
  Mission Impossible 3
 </title>
 ...
</listing>
```

Now, in order to detect changes in the content of the first document we define a trigger called new-movie that detects insertions of <movie> elements and act accordingly by inserting, for every new movie, a <title> element in the second file. The trigger definition is shown in Figure 4.3.

```
CREATE TRIGGER new-movie
WITH PRIORITY 1
AFTER INSERT OF document("movies.xml")//movie
FOR EACH NODE
 LET $newMovieTitle := NEW_NODE/title
 WHEN NEW_NODE/year/text() = '2006'
 DO ( FOR $listing IN document("listing.xml")//listing
      UPDATE $listing
      {INSERT <title>$newMovieTitle/text()</title>})
```

Figure 4.3: Definition of an Active XQuery trigger

4.2.2 Underlying Update Model and Language

As we mentioned before, Active XQuery assumes the existence of an update model that provides a data manipulation language for expressing updates on XML repositories. The event language defined here matches the underlying update language, and allows in this way the definition of events that reflect the effects of update operations. Active XQuery uses an Update language proposed in [70]. This language extends the XQuery language with primitives for supporting updates on XML repositories. The structure of updates follows the syntax given in Figure 4.4 on the following page:

```
FOR $bindings IN XPathExpression, ...
LET $binding := XPathExpression, ...
WHERE predicate1,...
UPDATE $binds {subOP {,subOP}*}
Where subOP is defined as follows:
DELETE $child
RENAME $child TO name
INSERT content [BEFORE | AFTER $child]
REPLACE $child WITH $name
FOR $binding IN XPathExpression,...
```

Figure 4.4: Definition of an Active XQuery trigger

4.3 XChange Active Language

XChange [11, 24, 38, 63] is a high-level, rule-based, active language for programming reactive behaviour and distributed applications on the (Semantic) Web.

One of the most important aspects of the language is the distinction between *volatile data* (event data) and *persistent data* (web data). Volatile data refers to information of the events that occur in the web, whereas persistent data refers to data stored at the database level (XML, RDF or any other database model). Moreover, volatile data can not be modified and it is best communicated in a pull manner. In contrast, web data can be updated and is communicated in a push manner. In order to deal with both, persistent and volatile data, XChange embeds an *event language* and a *query language*. On one hand, XChange uses the web query language Xcerpt [64] to access persistent data and express the condition part of ECA rules. On the other hand, an event language based on Xcerpt, is used for querying volatile data and implementing the event part of rules. For additional information regarding the query language Xcerpt see [64].

XChange uses the well-known Event-Condition-Action rules (ECA) paradigm to describe the reactive behaviour of applications. In XChange, a reactive program is a set of active rules. An XChange-enabled rule engine processes incoming events by evaluating event queries (expressed using the embedded event language) on the stream of events. If the evaluation produces a result (an answer), an event has been detected and the rule is fired. The next step in rule execution is to evaluate the condition part of a rule. In XChange this is done by evaluating a web query (expressed using the embedded query language). If the query evaluation produces an answer, the rule's condition is true and the action part is executed. As we see, XChange relies on the existence of an event language, a web query language and additionally, an update language for actions. Moreover, the three (sub)languages are fixed in XChange, which means that users have no choice but to use these (sub)languages.

The communication among XChange programs (located at different sites) follows the P2P model, where peers in a network can communicate with each other and no

centralized control is needed. Moreover, events are propagated in the Web following a *push strategy*. This means that after detecting an event, a web site communicates the event (event data) to all interested web sites.

4.3.1 Event Model and Event Messages

An event is something that occurs (a happening) in a web site at a point in time. In XChange, events can be low-level events such as an *update to an XML/RDF repository* or high-level, application dependent events such as *the cancellation of a flight from Lisbon to New York* in the Web of travel agencies.

Events in XChange are classified into two classes: *atomic events* and *composite events*. An atomic event is defined as before, i.e a happening or something that occurs in a system (web site). A composite event or situation is a combination of atomic and composite events. Atomic events can be further classified into two (sub)classes: *implicit events* and *explicit events*. Implicit events are local events, i.e internal happenings in a web site. They occur locally at a node and reflect local updates, changes in the system's state or queries executed on a local data source. On the other hand, explicit events are events that occur at some node in the web and are sent to another node (or posted internally at the same node). In this sense, explicit events are implicit events propagated from one node to another (possibly the same). For example, consider two web sites *A* and *B*. The insertion of a tuple in a local database at web site *A* is regarded as implicit event at *A*. It occurs inside *A* and it is processed locally at *A*. But, if as a result of the insertion operation in *A*, a reactive program sends a message to *B* with information about the occurred event (event data), the event is considered explicit at node *B*. Furthermore, atomic and composite events differ from each other in what regards to occurrence time. Atomic events have a single occurrence time, which in XChange is their reception time. Instead, composite events have a duration; i.e. they have a beginning time and an ending time. The beginning time is the reception time of the first constituent event that occurs while the ending time is the reception time of the last constituent event that occurs.

XChange uses *event messages* for representing and communicating events between web sites (possibly the same). An event message is an XML document that defines an envelope for an arbitrary XML content. The envelope contains information regarding the message's id, the sender and receipt of the message, as well as the event's occurrence and reception time. The arbitrary content is any valid XML document representing the event-specific information. In other words, event messages contain information about events that have occurred somewhere in the web. Using event messages, XChange-enabled web sites are capable of sending and receiving events. Let us consider an example [1].

[1]In all the examples we use the declaration xmlns:xch="http://pms.ifi.lmu.de/xchange"

Example 4.2 ***(Event message).*** *Consider an XML repository at Amazon.com, containing information about books. For each book, the web site stores the book's authors, the book's title and its price. In this example, an event could be the insertion of a new book. After the event has been detected, an event message containing information about the new available book is sent to every interested web site. Figure 4.5 depicts the structure of the event message (using the term-based representation of event queries and event messages [64]).*

```
xch:event{
  xch:sender {"http://www.amazon.com"},
  xch:recipient {"http://www.tagni.com.ar"},
  xch:raising-time {"2006-07-10T19:01:00"},
  xch:id {"1"},
  book {
    title {"The Jordan Rules"},
    author {"Sam Smith"},
    price {"19.90"}
  }
}
```

Figure 4.5: Event message containing information about the availability of a new book.

An event message's structure conforms to the following DTD.

```
<!DOCTYPE xchange:event [
  <!ELEMENT xchange:event (xchange:sender,xchange:recipient,
                           xchange:raising-time,xchange:reception-time)
                           xchange:id, %event-data>
  <!ATTLIST xchange:event xmlns:xchange CDATA #FIXED
                          "http://xcerpt.org/xchange">
  <!ELEMENT xchange:sender (#PCDATA)>
  <!ELEMENT xchange:recipient (#PCDATA)>
  <!ELEMENT xchange:raising-time (#PCDATA)>
  <!ELEMENT xchange:reception-time (#PCDATA)>
  <!ELEMENT xchange:id (#PCDATA)>
]>
```

4.3.2 Event Queries

In the same way event algebras use event expressions to define events (primitive/atomic or composite events), XChange uses a concept called *event queries*. Event queries are queries against event data. More specifically, an event query specifies or describes a pattern for the event representation (event message) to be queried. In XChange, event queries are used for event detection and data extraction. Events

(atomic or composite) are detected when event queries are evaluated against an incoming stream of events, i.e a sequence of event messages. Data extraction is achieved by specifying variables in event queries. Event queries variables are instantiated during query evaluation and their values are part of the answer. Moreover, multiple occurrences of the same variable act as join variables, in the same way as logical variables in Logic Programming.

In order to detect atomic and composite events the language provides *atomic* and *composite event queries* respectively. In XChange, an atomic event is an answer to an atomic event query, whereas a composite event is an answer to a composite event query.

Atomic event queries

An atomic event query describes or specifies a pattern for the representation of a single incoming event (event message). Every atomic event query is an *Xcerpt query term* [24, 64] with an optional absolute temporal restriction specification. Temporal restrictions are used for filtering the sequence of incoming event instances. Event instances whose reception time does not satisfy the temporal restrictions are discarded.

Example 4.3 ***(Atomic event query).*** *In order to detect an incoming event message like the one defined in the previous example, we could define the atomic event query showed in Figure 4.6. When an event is detected, variable* `Title` *will be instantiated with the book's title. The keyword before introduces the absolute temporal restriction.*

```
xch:event {{
  xch:sender {http://www.amazon.com},
  book {{
   title { var Title}
  }}
}} before 2006-07-31T23:59:59
```

Figure 4.6: Atomic event query to detect the event message defined in the previous example.

Composite Event Queries

Composite event queries allow the detection of temporal combinations of atomic events, i.e composite events or situations. They are specified by a combination of atomic event queries and *event query constructs* (operators). Figure 4.7 on the next page illustrates the use of event queries and event messages in event detection.

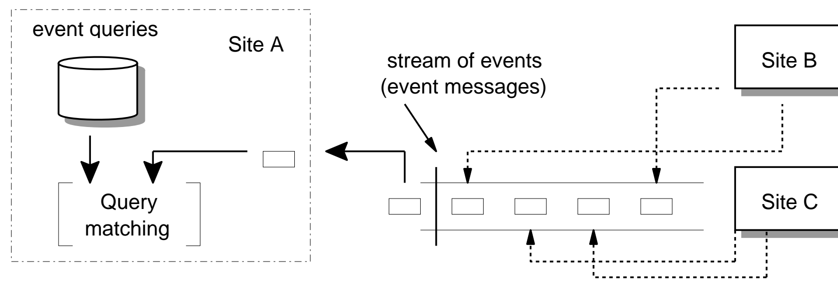

Figure 4.7: Using Event Queries and Event Messages in event detection.

Variables in XChange

Variables in XChange are treated in the same way as logical variables in Logic Programming, i.e they act as place holders for data. The language allows the definition or declaration of variables inside event queries, in which case they are used for retrieving data items (sub terms) from queried data terms. Variables can also be declared outside atomic event queries and in this case, they are bound to atomic events (to the answers of the event queries). Finally, variables in XChange can be defined outside a composite event query in which case they are bound to composite events (actually, they are bound to answers of composite event queries).

XChange also uses variables to implement the communication among the components of an ECA rule. In this case, variable bindings computed during event evaluation can be used in the condition or action parts of a rule, restricting in this way the possible bindings for the free variables appearing in these parts (same as in deductive rules in Prolog).

Event Query Constructs

The set of event query constructs provided by XChange includes *absolute* and *relative temporal restrictions*, *conjunctions*, *temporally ordered conjunctions*, *inclusive disjunctions*, *exclusions*, *quantifications* and *multiple inclusions and exclusions*. For a complete description of the event query constructs and their declarative semantics the reader is referred to [63].

- *Temporal restrictions* Incoming event instances (event messages) can be filtered out by defining temporal restrictions on them. For example, a Web site may decide to accept only those event instances whose reception time is less than a given value. Event messages that do not satisfy the imposed temporal conditions are discarded. XChange provides two types of temporal restrictions: *absolute* and *relative*. An *absolute temporal restriction* restricts an event reception time to an absolute time interval or point in time. Absolute temporal restrictions are defined using the keywords `before` and `in`. For instance, the event query defined in the previous example uses an absolute temporal restriction. On the other hand, a

relative temporal restriction restricts an event reception time to a relative interval. They are defined using the keyword `within`.

- *Temporally ordered conjunctions* A temporarily ordered conjunction specifies a composite event where the order of occurrence of its component events is of importance. That is, they impose a restriction on the order in which constituent events occur. XChange provides two variants of this operator. The first one is a *non-cumulative* version where an answer to such event query contains only instances of the constituent event queries. All other event instances are discarded. The second one, a *cumulative variant*, regards a valid answer as the one that contains, besides the instances that match the constituent queries, all event instances that occur in-between. An example of both types of queries is given below. Every time an answer to the composite event query is found, variables `Title` and `Artist` are instantiated. The non-cumulative version is defined as follows:

```
andthen [
  book {{ author {var Title} }},
  newCD {{ artist {var Artist} }}
]
```

  A cumulative version of the same event query is defined as follows (note the use of double angle brackets to indicate that the order of the answers is relevant):

```
andthen [[
  book {{ author {var Title} }},
  newCD {{ artist {var Artist} }}
]]
```

- *Exclusions* An exclusion query specifies that no instance of the excluded query should occur in a given time interval in order to detect the exclusion query. Exclusions event queries require the specification of a detection window, which can be defined by either a finite time interval or a composite event query (remember that composite events have a duration). Exclusions queries are defined using the pair of keywords `without` and `during`. The following exclusion query detects the non-occurrence of books notifications during the interval defined the two dates. Note that this type of event queries is evaluated at the end of the finite interval.

```
without {
  book {{ author {"Borges"} }}
} during [2006-07-10T23:59:59 .. 2006-07-31T23:59:59]
```

Evaluation of exclusion event queries is performed at the end of the time interval. In the example above, if at the of the interval (i.e on 2006-07-31T23:59:59) no notification of a book whose author is *"Borges"* has been received, the exclusion event occurs.

- *Quantifications* Using quantifications in event queries we can restrict the number of instances of a quantification event query in a given interval. Again, the interval can be specified by a finite time interval or a composite event query. For example, we can define an event query that detects `at least`, `at most` or `exactly` *N* instances of an event. The following event query detects an event when exactly 2 notifications of a new book are received at a web site during the given interval.

```
atleast 2 {
  book {{ author {var Author} }}
} during [2006-07-10T23:59:59 .. 2006-07-31T23:59:59]
```

XChange also provides a method for defining existential quantified variables. Variables not declared as existentially quantified require equality when occurring more than once in an event query. For example, the previous event query is successfully evaluated only if the two required answers produce the same binding for variable *Author*. Alternatively, if a variable is declared as existentially quantified this restriction is dropped and every successful evaluation of the vent query may produce a different binding for the variable. Variables declared as existentially quantified are defined outside the event query where they are used. An event query using an existential quantified variable is defined as follows:

```
atleast 2 any var Author{
  book {{ author {var Author} }}
} during [2006-07-10T23:59:59 .. 2006-07-31T23:59:59]
```

- *Multiple inclusions and exclusions* Multiple inclusions and exclusions are used for detecting occurrences of *N* event queries out of *M* defined event queries. It is used for expressing a generalized exclusive disjunction of event queries. As with the previous constructs, an interval must be specified. The following event query detects two occurrences of any of the event queries specified during the given interval.

```
2 of {
  book {{ author {"Borges"} }},
  book {{ author {"Cortazar"} }},
  book {{ author {"Allende"} }},
} during [2006-07-10T23:59:59 .. 2006-07-31T23:59:59]
```

4.4 A General Language for Reactivity in the Semantic Web

The reactive language proposed in [7] is also based on the ECA paradigm but, contrary to previous active languages, it allows for the integration and combination of different (sub)languages for specifying the components of active rules. The language is defined by means of an ontology that identifies its key concepts.

The language is based on an underlying model of the Web where, instead of a set of interconnected data sources we have a set of autonomous *Information Systems*. These systems are capable of detecting events that occur at different (remote) locations, updating persistent data, communicating changes to other systems and reacting to (local or remote) events. These properties, together with the open and heterogeneous nature of this environment, call for a declarative language for specifying reactivity and ultimately, evolution in the Semantic Web.

Although some reactive languages have already been proposed (like those presented and studied in this work), they exhibit some important problems, specially regarding the heterogeneity of (sub) languages and the types of events and actions that can be specified with them. More specifically:

- Most of the existing reactive languages allow for the specification of a restricted set of actions. In most cases, only updates operations over XML or RDF repositories are permitted. However, the vision of the Semantic Web considered in [7] requires the ability to express more complex actions such as sending messages across the network or the combination of simpler actions. Here, languages for specifying combination of actions may be required.

- Existing language consider only simple events such as the results of updates operations over data sources. However, the properties outlined before call for more complex and possibly composite events. Here, only the active language XChange presented before considers composite events whose constituent events are generic events reflecting local or remote situations. Note that, although there are several approaches for composite event detection and event algebras for specifying composite events, none of these approaches are active languages. However, some of them could be extended and then embedded into an active language.

- Heterogeneity is an important aspect that should be considered in an active language for the (Semantic) Web. Existing approaches do not consider this aspect, specially heterogeneity at the sub language level (event, query and action languages). For example, XChange fixes the languages for each of the components of an ECA rule.

The aim of the reactive language proposed in [7] is to overcome these problems and allow the combination of different languages for events, conditions and actions. This in turn will provide the means for specifying more complex events and actions so as to implement reactivity and evolution in the (Semantic) Web.

In the following sections we describe the most important aspects of the language and in particular, we focus our attention in the event component of ECA rules. For a detailed description of the ECA general reactive language presented here the reader is referred to [7].

4.4.1 ECA Rules

ECA rules in the language are composed by an *event*, a *condition* and an *action* part. The condition part is optional, as it can be integrated with the event or action parts. Additionally, the language allows the definition of an optional fourth component that specifies the post conditions that must hold after the action part is executed. This leads to a variant of ECA rules called ECAP rules.

Each of the components in a rule may be specified using a different (sub)language, i.e. they may be described using different languages. For example, we could define an ECA rule where its event part is specified using an event algebra (e.g. SNOOP), its condition part uses a Web query language (e.g. XQuery [20] and the action part is specified using an update language such as XUpadate [77], XPathLog [73] or XML-RL [51]. Moreover, different rules in a rule set may use different (sub)languages for the specification of their parts. Thus, heterogeneity at the rule part level is achieved by associating each rule part with a language and by defining a mechanism for the communication among components of a rule.

Finally, rules in the language are represented in the (Semantic) Web by using an XML-based Markup Language for ECA Rules (ECA-ML). ECA-ML represents event, condition and action components and associates them with the particular language used for the specification of the rule component. In this way, a rule engine for this ECA general language processes each component according to the language specified in the component definition.

Figure 4.8 on the facing page depicts the ontology for ECA rules in the language.

4.4.2 Rule Components and Languages

Languages for specifying event, condition and action parts are constituted by *constructs* and *composers*. Constructs can be combined by using composers in the same way operands are combined by operators in arithmetic expressions. Constructs represent classes of atomic events, conditions or actions. For example, in the event algebra SNOOP we may conceive a basic construct for representing the atomic event that occurs when a new book is added to a database. This construct can be defined

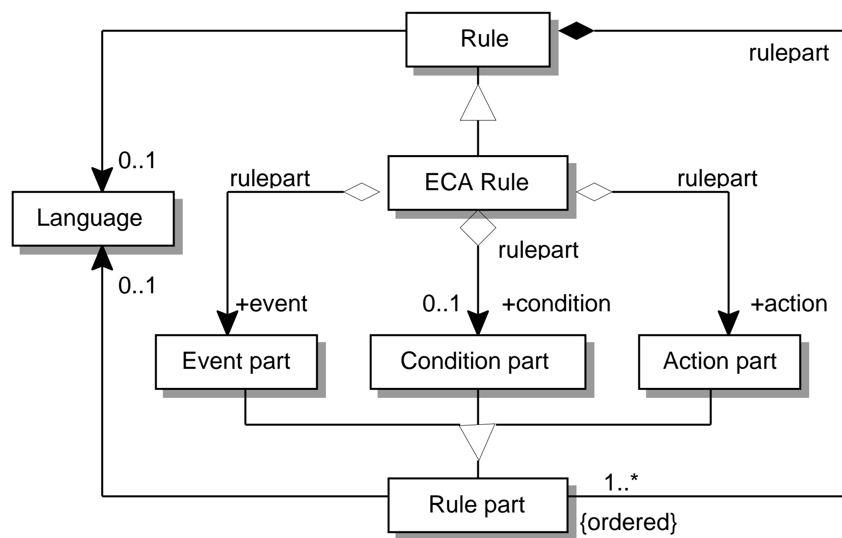

Figure 4.8: Rules an rule components.

as *newBook(BookTitle,BookPrice)*. In this case, *newBook* is the construct's name and both, *BookTitle* and *BookPrice* are its parameters. A construct's arity is the number of parameters it contains. Composers, on the other hand are used for combining constructs and other composers. The constructs to which a composer is applied are the composer's arguments and define its cardinality. Additionally, they can contain parameters. For example, the operator (composer) *ANY* in SNOOP can be used to define a composite event like *ANY(2,E1,E2,E3)*. Here, the number *2* is the composer's parameter and *E1,E2,E3* are the composer's arguments.

Rule components can be specified in different ways:

- *Opaque expressions* An event, condition or action component can be specified by an *opaque expression*. Opaque expressions are expressions written in some language (e.g. Prolog, Java, etc.) and associated with a set of logical variables. For example, the action component of a rule may contain an opaque expression written in Java that, when executed (evaluated) sends a notification by e-mail.

- *Constructs and Composers* By using the set of constructs and composers of the (sub)language associated with a rule component, it is possible to define an expression that specifies such rule component. For example, event expressions of the SNOOP event algebra can be used for specifying the event part of rules. Note that such an expression must be marked up using some sort of markup language (e.g XML).

- *Composition of specifications* Another way to specify a rule component is by combining several other specifications using different languages.

4.4.3 Interaction among Rule Components. Logical Variables

ECA rules may define and contain references to variables (logical variables). They act as place holders for data and can be instantiated (bound) by any rule component and used (referenced) in different parts of a rule. Rule parts communicate and exchange information with each other by means of logical variables, thus achieving heterogeneity at the rule part level. Moreover, variables can be shared among different rules in a rule set. In general, variables can be declared:

- *Locally at a rule part* Variables can be defined inside a rule part acting as a local variable for the component.
- *Globally at rule level* Alternatively, a variable can be defined at the rule level and hence, shared among the components of the rule.
- *Globally at rule set level* If we consider a set of ECA rules, a variable can also be defined at the rule set level. Here, the variable is shared among all the rules in the set, thus acting as a constant.

Figure 4.9 on the next page shows an illustrative example of an ECA rule defined in the language [2]. In this example, variables `Author` and `To` are declared globally to the rule, while variable `ValidYear` is declared locally at the condition component.

[2]Namespace declaration has been omitted for simplicity

```
<eca:rule>
 <eca:variable name="Author" value="Jorge Luis Borges"/>
 <eca:variable name="To" value="email@server.com"/>
 <eca:event language="http://www.snoop.org">
  <eca:variable name="Title"/>
  <eca:specification>
   newBook(Author,Title,Year)
  </eca:specification>
 </eca:event>
 <eca:condition language="http://www.languages.org/conditionLanguage">
  <eca:specification>
   <eca:variable name="ValidYear">$Year = '2006'</eca:variable>
  </eca:specification>
 </eca:condition>
 <eca:action language="http://www.languages.org/actionLanguage">
  <eca:variable name="Title"/>
  <eca:specification>
  if ($ValidYear)
    sendEmail($To,$Title)
  </eca:specification>
 </eca:action>
</eca:rule>
```

Figure 4.9: Example of an ECA rule defined in the language.

Chapter 5

Comparative Framework

Contents

5.1 Definition of the Comparative Framework 54
5.2 RDFTL: RDF Triggering Language 55
5.3 Active XQuery 60
5.4 XChange Active Language 64
5.5 Evaluation Results 69

The implementation of active languages requires, among other things, the implementation of ECA rule engines for processing active rules of the language. A key component of such systems is an event detector that implements the event language used in the active languages and hence, it detects events specified by expressions of the event language. This chapter is devoted to the study of those event detectors that have been implemented or proposed for the active languages considered in this work. We based our study on a comparative framework that we propose for comparing the prototype implementations. We proceed by first defining the criteria that we have considered in order to evaluate the systems. After that, sections 5.2 to 5.4 describe the architectures and prototypes for each of the active languages. Then, in section 5.5 we present the evaluation results.

5.1 Definition of the Comparative Framework

The active languages introduced in chapter 4 are based on the ECA paradigm. As we have already mentioned, these languages allow the specification of active rules and each rule comprises three components, namely an *event part*, a *condition part* and an *action part*. In order to specify which is the event that fires a rule, the condition that must hold and the action to be taken upon event detection, a (sub)language for each of these components is needed. In other words, each of the components is specified by a (sub)language. As we have seen, an event language allows the definition of event expressions, each of which denotes a composite or atomic event. Finally, in order to actually detect the events specified by event expressions what we need is an *event detector*.

As we have already described in chapter 3, several systems have been developed to provide passive databases with reactive capabilities and, in recent years, interest for reactive behaviour has shifted from the database community to the Web community. In the vision of the Web presented in [7], autonomous systems interact with each other exchanging information and changes (events). The exchange of events reflecting changes in the nodes of the Web is a key aspect in the implementation of reactive applications in the Web. Moreover, the interaction among event detectors is also important as they can detect remote events that occur globally in the Web. For example, imagine a situation where a reactive system (Web site) uses an event detector to detect composite events about book selling. This event detector may need to contact (interact with) another detectors in order to perceive atomic events. Furthermore, depending on the type of atomic events an event detector is interested in, the atomic event provider may be different. This scenario requires an event detector to be able to interact with possibly multiple event detectors, exchanging not only event instances but also event definitions.

The comparative framework that we propose in this work aims at studying and comparing the use of event detectors in the context of active languages. We want to analyze the interaction between event detectors and systems implementing a particular active language. We intent to study different event detectors so as to identify the key aspects of their implementation and in this way, formulate guidelines for the implementation of event detectors in general, and in particular, for the implementation of an event engine to be integrated in the ECA framework [7]. These guidelines may suggest the use of a particular technique for detecting events and also, allow the improvement of existing approaches. We also pretend to investigate the possibility of reusing existing event detectors.

The key aspects that we consider in our comparative framework are the following:

- *Technique used for detecting events.* If the active language provides an implementation of an event detector, we want to study the technique used for detecting

events. Otherwise, based on the characteristics of the event expressions and a possible architecture for a rule engine, we want to investigate which technique could be used.

- *Types of events.* A taxonomy of events is more than important as the type of events involved in a system determines the techniques to be used. For example, detection of events in the context of XML or RDF repositories may require different techniques. In the same way, detection of global events in the (Semantic) Web may require a combination of techniques. Moreover, atomic and composite events are detected using different approaches.

- *Event parameters.* Here we want to focus our attention on how the event parameters are handled by an event detector. This includes how event parameters are computed and communicated to other rule-parts or remote event detectors.

- *Communication of atomic events.* A key aspect for composite event detectors is the perception of atomic events. Thus, studying how atomic events are provided to composite event detectors is very important.

- *Integration and reuse.* We want to investigate the possibilities of reusing existing event detectors in order to detect events expressed in each of the active languages.

5.2 RDFTL: RDF Triggering Language

The implementation of a system supporting RDFTL ECA rules in centralized and distributed environments is presented in [57–60]. The system is implemented as a set of services that together provide a passive RDF repository with active capabilities. The main component of the system is an *ECA Rule engine*, which coordinates the execution of RDFTL ECA rules. *Event Detection*, *Condition Evaluation* and *Action Execution* services are provided by different components that interact with the ECA engine. The system assumes the existence of an RDF repository capable of providing *Query* and *Update* services. This means that the updates operations executed in rules' actions are actually data manipulation operations provided by the underlying Update service. The query service is used for evaluating RDFTL Path expressions. Before continuing, it is worth mentioning that the execution of updates follows the same approach used in [59] for ECA rules on XML repositories. According to this, when an update operation is executed, the system annotates the inserted fragments in the RDF graph (for an insert operation) or, it annotates the deleted fragments without actually deleting them (for a delete operation); the delete operation is postponed until the end.

In this section we briefly describe the system architecture for both centralized and distributed environments, the rule registration and execution mechanisms, as well as the general aspects of the system. In particular, we are interested in analyzing the

event detector used in the system according to the different criteria of our comparative framework. We first describe the prototype for supporting RDFTL ECA rules in distributed environments. Then, we proceed to describe a prototype implementation for centralized environments.

5.2.1 Supporting RDFTL ECA Rules in Distributed Environments

In the distributed approach, RDF/S descriptions are distributed across a set of passive RDF repositories. Each RDF repository is extended by a set of services providing the passive repository with active capabilities. The system architecture for distributed environments comprises two types of nodes: *peers* and *super-peers*. RDF/S descriptions are distributed across the peers (sites) of a network by storing fragments of them at different peers. Each peer has a local RDF repository which stores a fragment of the global RDF/S descriptions. This is similar to the data distribution approach in distributed databases, where portions of a table are distributed across different sites of a network. Each super-peer supervises a set of peers and stores a fragment of the global RDF/S descriptions. For example, an RDF document containing information about books and movies may be fragmented over two peers, both perhaps being controlled by the same super-peer. We could store the information about books in one peer and the information about movies in the other peer. Figure 5.1 depicts the architecture for supporting RDFTL ECA rules on distributed environments. An *ECA Engine* includes an *Event Handler (EH)*, a *Condition Evaluator (CE)*, a *Rule Base (RBase)*, an *Action Scheduler (AS)* and an *Execution Scheduler*.

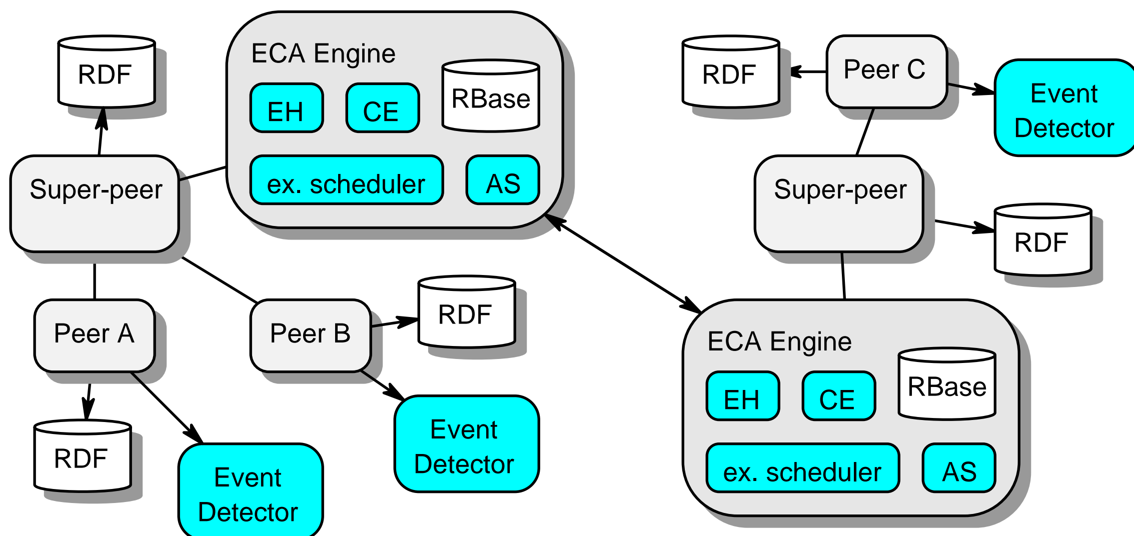

Figure 5.1: System Architecture for Distributed Environments

Besides storing fragments of RDF descriptions, peers and super-peers store fragments of a global RDFS schema. This information is used for supporting registration, propagation and processing of ECA rules in the system, as well as evaluation of conditions. Additionally, each super-peer has a rule base where RDFTL ECA rules are stored

using an XML format. Rules in the system might be triggered, considered (evaluated) and executed at different sites of the network. However, distributed event detection and update execution is not yet supported by the system. That is, rules stored at a super-peer's rule base can only be triggered by events occurring within a single peer's local RDF repository. In the same way, actions of a rule are executed at a single local repository, although different (sub)actions of a rule may be executed at different peers. For example, if a rule's action part contains two actions, one of them could be executed at one peer's local repository and the other at another peer's local repository. Only conditions can be evaluated at different sites.

Furthermore, each super-peer contains an *ECA Rule Engine* implemented as a Web Service that provides a passive RDF repository with active capabilities. The rule engine comprises several components including an *Event Handler*, a *Condition Evaluator*, an *Action Scheduler* and an *Execution Schedule*. In addition, each peer in the network contains an *Event Detector* whose functionality is complemented by the *Event Handler* in the *ECA Rule Engine*.

- *Event handler* The event handler is responsible for receiving and processing notification from the event detector about the occurrences of events. Whenever a change to the local RDF repository in a peer (in the SP's peers group) is performed, the event detector detects the event and notifies the event handler. Then, the event handler tries to determine which rule in the rule base should be triggered. Note that the event handler *does not* detect events. It just implements triggering of rules based on the occurred events.

- *Condition evaluator* This component evaluates the condition of the triggered rules. It does this by sending a query to relevant peers. If the condition is true, the rule is fired or executed.

- *Action scheduler* This component is responsible for selecting the actions from the fired rules that must be executed.

- *Execution schedule* This a repository for update operations. Each update operation belongs to the underlying update language and does not contain reference to any variable.

- *Event detector* This component is responsible for event detection in the system. We will see this component in details later in this section.

Rule Registration and Execution

Whenever a rule is registered at a peer it is sent to its super-peer for storage. Then, the super-peer sends the rule to all the relevant super-peers for storage (using the information in the annotated RDFS schema). Every time a SP receives a rule from one of

its peer it annotates the rule with the peer's ID. This information will be used later for determining triggering of rules. For additional information about the distribution of rules and how this is performed in the system, the reader is referred to [57,59,60].

When an update operation is executed at a peer, the corresponding super-peer is notified of any event that may have occurred. After this, the super-peer checks which are the rules that might have been triggered. If a rule could be triggered, the ECA engine at the super-peer evaluates the event query (RDFTL Path expression) of the rule's event part. If the result of this evaluation (a set of nodes) contains any annotated nodes, then the rule is triggered and the ECA engine proceeds to evaluate the condition. At this point, the ECA engine generates an instance of the condition for each possible value of the system variable `$delta` (instantiated by the event detector), provided the variable is referenced in the condition part. If the condition is evaluated to true, the super-peer sends each instance of the action part (depending on whether or not the variable `$delta` is used in the action part) to the corresponding peers for execution (this is done by using the information of the annotated RDFS schema). Additionally, instances of the action part are also sent to the relevant super-peers for execution. In this way, execution of update operations may raise events and cause a local or remote action schedule to change.

Event Detection Service

In the distributed environment, each peer provides an *event detection service*. This service is implemented by a local event detector that detects changes in the peer's local RDF repository. Every time an event is detected, the event detector determines the event's occurrence time, the type of event and the portion of metadata affected by the event. After this, it notifies the event handler in the corresponding super-peer by sending this information.

The technique used for detecting events is based on the monitoring of updates operations. An event detector is implemented as a *wrapper* placed on top of an RDF repository. Every update request sent to the RDF repository passes through the wrapper and then is redirected to the repository. After the update operation is executed, the event detector determines the event type, the event's occurrence time and the affected fragment. After this, the event handler is notified by the event detector. Additionally, the event detector notifies its local *Peer Indexing service* so as to update its RDFS schema information in order to reflect the changes.

The event detector determines the event type based on the type of operation executed on the RDF repository. An event's occurrence time is the time at which the update operation is performed, while the affected RDF fragments are determined by evaluating the event's *RDFTL Path expression* on the RDF repository (using the *Query service* provided by the underlying repository).

5.2.2 Supporting RDFTL ECA Rules in Centralized Environments

In a centralized environment, an *ECA Rule engine* for processing RDFTL rules is installed on top of an RDF repository. The ECA Engine implements a set of services that, together with the repository's *Query and Update Service*, provides a passive RDF repository with active capabilities. Compared to the previous approach, in this centralized scenario there is no need for keeping annotated versions of an RDFS schema, as the RDF descriptions are not distributed. Therefore, registration and processing of rules is simpler than in the previous case.

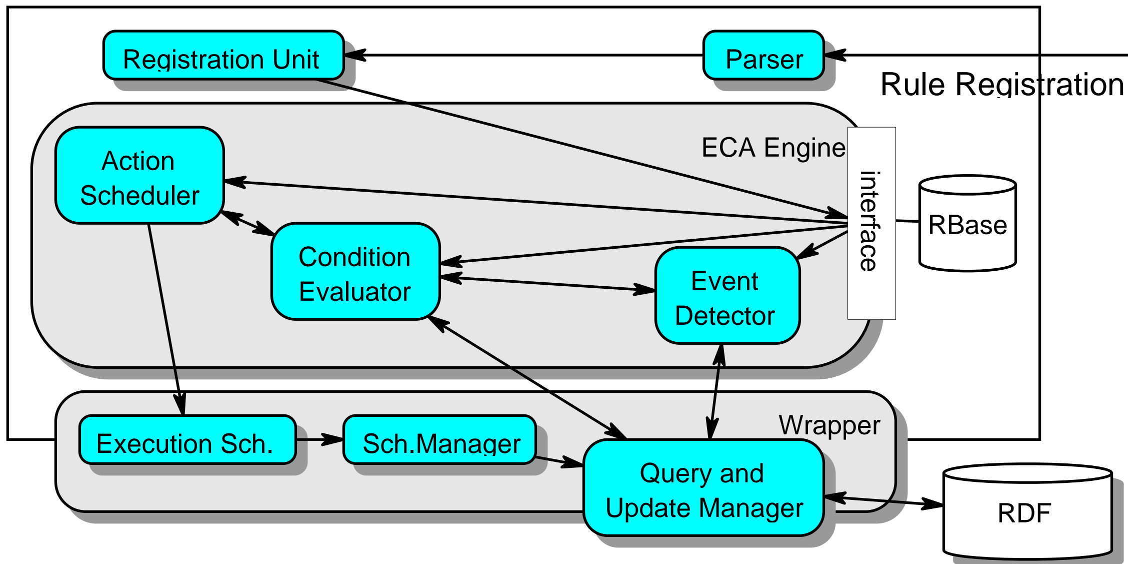

Figure 5.2: System Architecture for Centralized Environments

The architecture of the ECA Engine for the centralized scenario is illustrated in Figure 5.2 [1]. This architecture, with appropriate changes to manage XML documents, was previously used for implementing reactive capabilities on top of XML repositories [59]. The most important components of the system are:

- *Execution Schedule* This is a sequence of updates operations to be executed on the RDF repository. None of the operations contains a reference to the system variable `$delta`.

- *Schedule Manager* This component is responsible for selecting the next operation to be executed on the repository and sending an execution request to the repository's *Query and Update Service*.

- *Event Detector* This component is responsible for detecting changes in RDF documents stored in the RDF repository. Detection of events is implemented by monitoring update requests. Every update operation sent to the repository for execution passes through the *ECA Engine*. After the operation is executed, the *Schedule*

[1]The figure is a modified version of the one in [59]. It has been adapted to the RDF scenario

Manager passes the control to the *Event Detector*, which based on the type of operation determines the type of event that has occurred. Additionally, the event detector is responsible for determining for each rule that might have been triggered the set of affected nodes. This is done by submitting the rule's event expression to the repository's *Query and Update Service*. If the result contains any new or deleted node, then the rule must be triggered. The set of affected nodes for each triggered rule is stored in the system variable `changes`.

- *Condition Evaluator* After a rule is triggered, the *Condition Evaluator* submits the rule's condition part (path expression) to the repository's *Query Service* for evaluation. Here, if the variable `$delta` appears in the condition part, the evaluator creates an instance of the rule's condition part for each value of the `changes` variable. Each instance is then submitted for evaluation.
- *Action Scheduler* When a rule is fired, the action scheduler reformulates the action(s) in the rule's action part and sends the individual actions to the *Execution Schedule* for storage. Reformulation of actions consists, among other things, of replacing every reference to the variable `$delta` by the current node of the *delta* set (set of affected nodes).

In addition to these components, the system includes a *Registration Unit* that interacts with a *Parser* in order to allow registration of RDFTL ECA Rules and a *Rule Base* component for storing ECA Rules (as in the previous approach, rules are stored using a XML-based Markup language).

5.3 Active XQuery

The description of the language presented in chapter 4 made no mention to the trigger execution model. In this section, we describe the general aspects regarding the execution of triggers defined with the language. In particular, we want to focus our attention

Recall that the language allows the definition of active rules for implementing reactive functionalities on XML repositories. That is, rules in the language are used for expressing actions that must be taken when events (changes in the XML repository) occur, provided stated conditions hold. In this way application programs are able to monitor changes in XML documents and react accordingly to them. These changes are the result of updates operations executed on the XML repository.

5.3.1 Semantics and Execution Model

In Active XQuery, triggers are executed according to the *immediate execution model* [21]. This execution model processes the triggers immediately after their corresponding

events occur. In other words, rules are considered as soon as the events that fire them are detected in the system. This execution model contrast with the *deferred execution model* where rules are triggered based on an event history (also called edit-script) that keeps track of the changes made to XML documents. For example, under the immediate model, whenever an event occurs, the system selects those triggers whose event component matches the event and then fires the triggers. Under the deferred model, every event is stored in an event history. Later, the system inspects the event history and fires the corresponding triggers.

The language adapts the execution model of SQL3 triggers. However, due to the hierarchical structure of XML documents and the "bulk" nature of update statements, the SQL3 trigger execution model must be revised in order to be successfully adapted. More specifically, update instructions may affect an arbitrarily large fragment of XML (thus the term "bulk"). In other words, a single update statement may insert, delete or replace a large portion of XML content. For instance, in example 4.1 (section 4.2.1), the update instruction `UPDATE {INSERT $movie}`, where `$movie` is defined as `$movie IN document(movies.xml)//movies/movie`, inserts a whole subtree into the XML document. The main problem with bulk statements is that triggers whose event components refer to internal portions of the affected fragments are never triggered by update statements. Let us illustrate this with an example.

Example 5.1 ***Bulk statements and triggers.*** *Consider the XML document movies.xml introduced in the example 4.1 (section 4.2.1) and the* ***UPDATE*** *statement S1. The document newmovies.xml contains a list of new movies that will be added to the first document. Figure 5.3 on the following page depicts the tree representing the XML document obtained after the operation is executed. The dashed line represents the affected new fragment.*

```
S1: FOR $movies IN document("movies.xml")/movies
        $newMovie IN document("newmovies.xml")/new-movies/movie
    UPDATE $movies {INSERT $newMovie}


CREATE TRIGGER new-title
WITH PRIORITY 1
AFTER INSERT OF document("movies.xml")//title
FOR EACH NODE
  LET $newMovieTitle := NEW_NODE/title
  DO ( execute some action)
```

Now, let us suppose that the system contains a trigger called new-title that monitors the insertion of ***<title>*** *elements in movies.xml. When the update instruction inserts a new movie in the database, the system is unable to trigger this rule. This is because the trigger's event component does not match the update event. In other words, the insertion of* ***<title>*** *elements is hidden in the* ***INSERT*** *operation and can not be detected. Remember that in Active XQuery events correspond to update operations.*

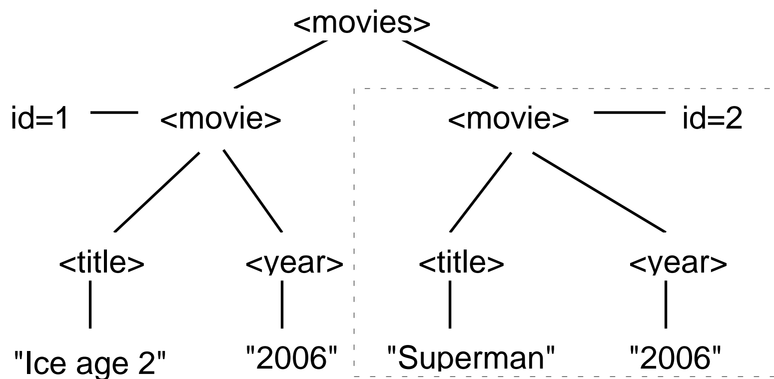

Figure 5.3: XML tree showing the result of executing a "bulk" insert

Expansion of update instructions

To overcome the problem with bulk statements, Active XQuery uses an expansion mechanism that decomposes bulk statements into a sequence of equivalent smaller granularity updates. These new updates, when executed on the repository, produce the same results as the original update operation. Each of these smaller updates can be thought of as an autonomous update primitive that modifies a portion of the XML content. The expansion mechanism makes triggers sensitive to events that occur when portions of the affected fragments are changed.

The strategy, which implements the so-called *loosely bundling semantics* [21], receives as input a bulk statement and produces a sequence of updates interleaved with directives to the rule engine. These directives are used by the rule engine to execute *after* and *before* triggers in combination with the generated update primitives. The decomposition algorithm proceeds with a combined depth-first and breadth-first visit of the involved fragments. All the insert operations related to XML nodes with the same father are performed at the same time, i.e encapsulated in the same `UPDATE{}` operation. Attribute nodes are inserted first. Nodes with a complex structure are further decomposed. Finally, nodes with PCDATA content are inserted together with their content. Delete operations can be treated similarly, while replace operations can be simulated with a combination of delete and insert operations.

Example 5.2 ***Decomposition-based strategy.*** *Consider the update statement S1 introduced in example 5.1. Its decomposition generates the following two smaller updates S2 and S3.*

```
evalBefore(S2)
name: S2
FOR $x IN document("movies.xml")/movies,
    $newMovie IN document("newmovies.xml")/new-movies/movie
UPDATE $x {INSERT </movie>}

evalBefore(S3)
```

```
name: S3
FOR $x IN document("movies.xml")/movies,
    $newMovie IN document("newmovies.xml")/new-movies/movie,
    $curfragment IN $x/*[empty($x/*[AFTER $curfragment])]
UPDATE $curfragment {INSERT new_attribute(id,"2")
                     INSERT <title>Superman</title>
                     INSERT <year>2006</year>
                    }
evalAfter(S3)
evalAfter(S2)
```

*The directive **evalAfter(S3)** tells the rule engine to compute the set of after triggers triggered by the execution of the update S3. **evalBefore(S2)** is used in the same way but, with before triggers.*

Executing triggers and update instructions

Execution of triggers and update statements proceeds as follows. When an update statement is sent to the system, the *Decomposition Module* decomposes the bulk statement into a sequence of smaller updates (also called expanded statements) and directives to the rule engine (*EvalBefore* and EvalAfter directives). Before the decomposition process takes place, the set of relevant fragments is computed by submitting a query to the *Query Engine*. Relevant fragments are used for decomposing the bulk statements and for instantiating the transition variables `NEW_RF` and `OLD_RF`; used for exchanging data among the components of a rule. In other words, relevant fragments specify the fragment of data to be inserted, deleted or updated and where, in the target XML document, this fragment is or will be.

After the relevant fragments (RF) and the sequence of expanded statements and directives (SIL) are computed, the *Rule Engine* starts an iterative process to execute each of the elements in SIL. If an $EvalBefore(S_n)$ directive is found, the rule engine selects from the rule repository the *before triggers* activated by the expanded update S_n. If an $EvalAfter(S_n)$ directive is found, the rule engine selects the *after triggers* activated by S_n. In both cases, the set of activated triggers is processed by the rule engine. Finally, if the element in SIL is an expanded update, the rule engine executes it.

When a trigger is processed, the rule engine computes the transition variables, evaluates the condition and, if the condition is true, the rule engine executes the trigger. For a detailed explanation of the triggers execution model and pseudo-code of the execution algorithm the reader is referred to [10,21].

The proposed architecture for a prototype implementation is depicted in Figure 5.4 on the next page. The main components of the system for executing Active XQuery rules are the *Decomposition Module*, the *Rule Engine* and the *Query Engine*. Addition-

ally, the architecture comprises a *Rule base* and an *XML repository*. Event detection, condition evaluation and action execution are performed inside the rule engine.

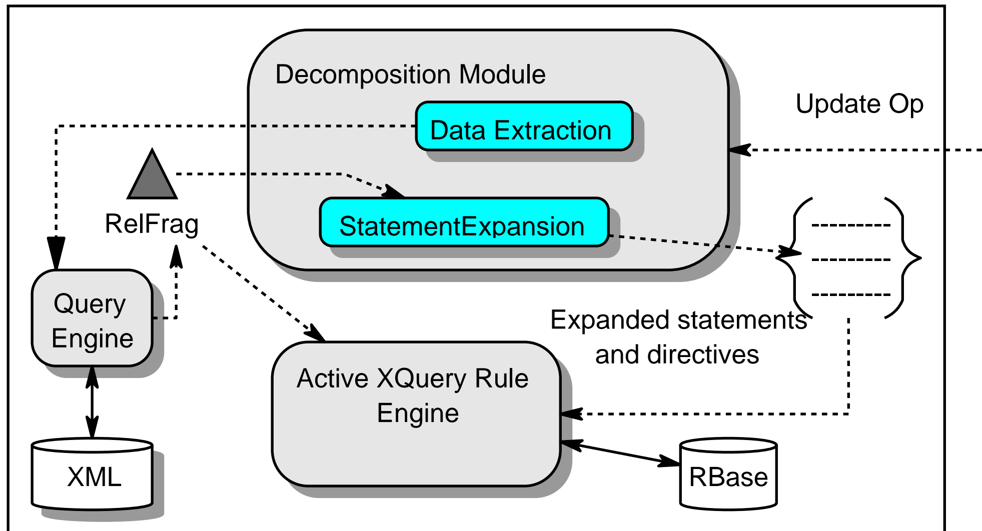

Figure 5.4: Architecture of a Active XQuery Rule System

5.4 XChange Active Language

In the previous chapter (section 4.3), we introduced the active language XChange without mentioning how the language is actually implemented. This section presents the prototype implementation of a system for processing XChange Active Rules. We start by describing the system architecture and then we move to the event detection functionality implemented by the system. In particular, we are interested in the technique used by the event detector for detecting events, the representation of event expressions, the processing of event parameters as well as any other important aspect that may serve for the evaluation of the event detector.

5.4.1 System Architecture

The architecture of a system implementing the language XChange, described in [38], is composed by three main components: an *Event Detector*, a *Condition Handler* and an *Action Handler*. The event detector is in turn composed by an *Event Receiver* and an *Event Handler*. Figure 5.5 on the facing page depicts the architecture of the prototype implementing the XChange Rule Engine.

- *Event Detector.* The event detector is the principal component in the system as it is responsible for detecting the events that trigger active rules. Event queries (event expressions) registered with the system denote the set of events to be detected by the event detector; they belong to the event language defined in XChange. The event detector detects events by evaluating registered event queries against

the incoming stream of atomic events (event messages). If a sequence of atomic events satisfies an event query (i.e., it is an answer for the event query) the event (composite or atomic) denoted by the event query is detected. Whenever an event is detected, the event detector notifies the condition handler; sending the event (composite or atomic) and the substitution set for the variables defined in the corresponding event query. The functionality of the event detector is implemented by two (sub)components, an *event receiver* and an *event handler*. The event receiver is responsible for dealing with incoming atomic events (event messages); it computes the event's id and reception time and then passes the atomic event to the event handler. The event handler, on the other hand, is responsible for detecting events and then notifying the occurrence of them to the condition handler.

- *Condition Handler.* This module is in charge of evaluating the conditions of those active rules that were triggered by the occurrence of an event. The condition handler, through the condition channel, receives notifications of successful event query evaluations together with the corresponding substitution set. Variables in the substitution set are used for evaluating the condition part of the triggered rules.

- *Action Handler.* This component is responsible for the execution of rule actions. It receives a notification from the condition handler about the evaluation of a rule's condition and then proceeds to execute the list of actions.

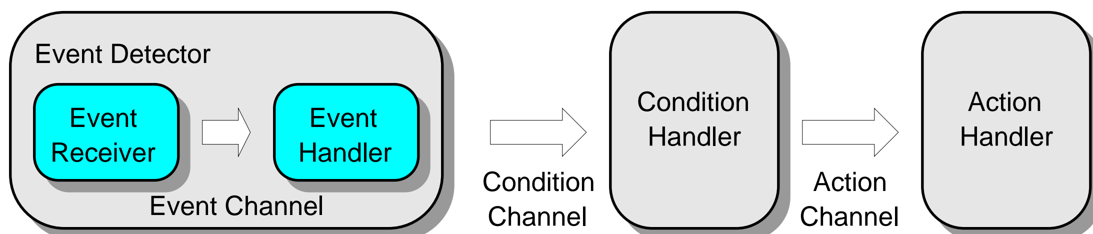

Figure 5.5: Architecture of a prototype Rule Engine for XChange active rules

The communication among components of the system is implemented using *communication channels*; following the First-in, First-out (FIFO) communication model. These communication channels are used for exchanging event messages and information about the evaluation of rules conditions. The system implements three communication channels, the *Event Channel*, the *Condition Channel* and the *Action Channel*.

- *Event Channel.* This channel communicates incoming atomic events (event messages) to the event handler. It receives atomic event messages from the *event receiver* and forwards them to the *event handler*.

- *Condition Channel* The condition channel is used by the event detector to communicate successful event query evaluations to the *condition evaluator*; it comprises

the sequence of events that answered the event query together with the substitution set for the free variables in the query.

- *Action Channel* This channel is used by the *condition handler* to pass information about successful condition evaluations to the *action handler*.

5.4.2 Event Detection in XChange

Event detection in XChange is based on the evaluation of event queries. Every time an atomic event (its representation as event message) arrives at the system, the event detector must evaluate all the partial event queries registered with the system. Evaluation of event queries depends on the type of event query being evaluated, i.e. atomic or composite event queries. In order to evaluate event queries, the event handler considers the event query to be evaluated, the current atomic event and, in case of a composite event query, some of the previously received atomic events; which are stored in the system. The result of evaluating an event query is an answer to the event query; once again, the answer depends on the type of event query being considered. In any case, if the evaluation process is successful the event handler obtains a modified partial event query and a list of composite events [2]. Whenever an event is detected the event handler notifies this to the condition handler, which in turn initiates the condition evaluation process of the triggered rules. During the event query evaluation process atomic events whose life-span has expired are discarded.

- *Answers to atomic event queries.* Answers to atomic event queries are atomic events; represented as event messages (XML documents). The atomic event must satisfy the temporal restrictions that might be defined in the event query. Answers to atomic event queries consist of the single atomic event that matches the event query's structure and a *maximal substitution set* for the free variables defined in the event query.

- *Answers to composite event queries.* An answer to a composite event query is a composite event. For composite event queries, answers consist of the sequence of atomic events used for answering the constituent event queries and a *maximal substitution set* for the free variables defined in the event query. Additionally, a composite event query answer includes the the beginning and ending time of the composite event.

Evaluation of Atomic Event Queries

In order to evaluate an atomic event query, the event detector tries to match the event query with a single incoming event message. This process is implemented using an method called *simulation unification* [25]. Basically, what this method tries to do is to

[2] note that an atomic event is a special case of a composite event with only one constituent event

find the event query's structure in the event representation. If the event query and the current event match, the evaluation process returns an answer to the event query; otherwise, no answer can be computed.

Evaluation of Composite Event Queries

Evaluation of composite event queries is used for detecting composite events. A composite event is detected when the event query specifying the event is answered by a sequence of atomic events (a combination of them). More specifically, composite event queries are evaluated in an incremental manner. This means that every time an answer to a composite event query's constituent query is found, the event detector saves the information about the atomic event used for computing the answer. Let us illustrate this with an example.

Example 5.3 ***Incremental evaluation of composite event queries.*** *Consider the composite event query depicted in Figure 5.6, which denotes an event that occurs when both a book written by "Borges" and a movie directed by "Spielberg" are available at an online marketplace. For simplicity, we write event queries without the envelope information. The figure also shows the sequence of atomic events received by the event detector at a given point in time.*

```
and {
  book {{ author {"Borges"} }},
  movie {{ director {"Spielberg"} }}
}

book{author{"Borges"}}, movie{director{"Spielberg"}}
```

Figure 5.6: A composite event query defined using the conjunctive operator.

After the first atomic event is received, the event detector finds an answer for the constituent event query ***book {{ author {"Borges"} }}****; this information is saved by the event detector. When the third atomic event is received, the event detector is able to answer the event query* ***movie {{ director {"Spielberg"} }}****. At this moment, the event detector knows that both constituent event queries were answered and thus, the composite event query is answered and the composite event has occurred. Note that using incremental evaluation the event detector is able to memorize the results of previous evaluations.*

Evaluation of Event Queries using Event Trees

Composite event queries registered with the system are internally represented using a tree-based approach. Leaf nodes implement atomic event queries, while inner nodes implement the event language's composers; an internal node represents a composite event whose constituent events are represented by the node's children. Figure 5.7 on

the following page illustrates the event tree associated with the event query shown in Figure 5.6 on the previous page.

In the prototype implementation, every time an atomic event arrives, the event handler evaluates every partial event query registered in the system. The process of evaluating composite event queries (i.e. detecting composite events) is implemented by a recursive function. The function takes as parameters the current atomic event and a partial event query. Its output is an updated partial event query together with a set of composite events. This function operates by recursively traversing the event tree representing the event query being evaluated. If the current node is an operator node, the function is applied to every child. After processing every child node, the function determines the composite events for the current operator node based on the constituent composite events obtained from each child. Every time the function processes a node, the set of returned events are sent to the node's parent. If the node is the root of the event tree, the event detector notifies the condition evaluator; this means that an XChange active rule must be triggered.

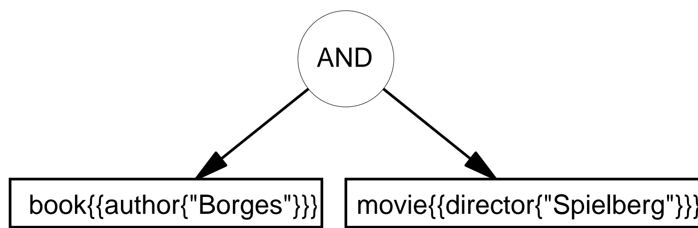

Figure 5.7: Event query represented using an event tree

5.4.3 Working with Logical Variables

As we mentioned at the beginning of this section, the result of evaluating an event query is an answer to the query. Besides the sequence of atomic events that match the event query's pattern, an answer includes a *maximal substitution set*. This set specifies the assignment of values for the free variables appearing in an event query; it is a set of pairs of the form *(variable,value)*.

During the evaluation of an event query, the event detector must compute, for every operator node in the event query's tree, the set of valid composite events. This is done by considering the constituent composite events returned by the operator's children. Additionally, the binding of variables appearing in the constituent event queries must be computed. Instead of using substitution sets, the prototype implementation works with constraints among variables. Intuitively, if a variable appears more than once in the same or different constituent event queries then they must have the same value. In order to implement this mechanism the prototype discussed here invokes a constraint solver provided by the language Xcerpt. Let us illustrate this with an example.

Example 5.4 ***Computing variable bindings.*** *Consider the composite event query de-*

*picted in Figure 5.8 and the stream of incoming atomic events **S1 = book{author{"Dan Brown"},title{"The Da Vinci Code"} }, movie{director{"Spielberg"},title{"The Da Vinci Code"}}**. The event detector matches the operator's left-hand side event query with the first event message; producing the binding {X = "The Da Vinci Code"}. This information, together with the atomic event just detected is propagated to the node's parent. When the second atomic event arrives, the event detector matches it with the second event query; producing in this case the binding {X = "The Da Vinci Code"}. After this, the event detector tries to compose a composite event for the `and`-operator by checking the constraints among variables. Since the value for the variable X is the same in both event queries, a composite event is detected with variable binding {X = "The Da Vinci Code"}. On the other hand, if the stream of atomic events is **S2 = book{author{"Dan Brown"},title{"The Da Vinci Code"} }, movie{director{"Spielberg"},title{"A.I."}}**, the constraints among variables will not be satisfied as {X = "The Da Vinci Code"} when the left event query is evaluated and {X = "A.I."} when the right query is evaluated.*

```
and {
  book {{ author {"Dan Brown"},title{var X} }},
  movie {{ director {"Spielberg"},title{var X} }}
}
```

Figure 5.8: A composite event query defined using the conjunctive operator.

5.5 Evaluation Results

In this section we present the results of the comparative analysis. We proceed by describing the proposed event detectors for each active language according to the criteria defined in our comparative framework.

5.5.1 RDFTL

Types of Events

RDFTL fixes the event language used for defining events. Detectable events reflect changes in RDF repositories caused by the execution of update operations. More specifically, we can distinguish between two classes of events. The first class is called *node-oriented events* and reflects changes to the nodes in the graph-based/triple representation of an RDF document. The second class of events is the so-called *triple-oriented events* which reflect modifications to RDF triples. Triples can be inserted, deleted or updated; the update of a triple changes the object value of the triple.

The event language does not allow the specification of composite events, i.e. only primitive events can be detected. Moreover, the type of primitive events that can be

specified (detected) is closely related to the type of update operations supported by the RDF repository. Furthermore, as no standard update language for RDF exist yet, if the RDF repository changes, the event language have to be modified so as to match up with the update language.

Technique for Detecting Events

In both distributed and centralized scenarios events are detected by monitoring the update operations sent to the RDF repository. The event detector acts as a *wrapper* that receives the update operations and then redirect them to the repository. Based on the type of operation the event engine is able to determine the type of primitive event. This approach is quite simple and effective and most importantly, it does not depend on the type of update operations supported by the RDF repository, i.e. everything sent to the repository is catched.

Although the technique for detecting events in both scenarios is the same, the functions performed by the event detector are not. In the distributed scenario, the event engine's work is complemented by an event handler. Event detectors are located at each peer, whereas event handlers are located at super peers. The event detector detects events and then notifies the event handler. The later is responsible for determining the triggered rules. On the other hand, in the centralized scenario, triggering of rules is performed by the event detector. Notice that the first approach is better than the second one as the task of an event detector should remain as simple as possible. That is, ideally an event detector should *only* detect events, compute the event parameters and the event's occurrence time. Then, another module should be responsible for determining which rule should be trigger. This approach keeps the event detector as simple as possible and promotes modularity, extensibility and optimization of tasks.

Event Parameters

Besides the event's occurrence time and the event type, event parameters include the affected portion of an RDF document, i.e. the set of RDF nodes or triples that were inserted, deleted or updated by a RDF repository operation. Event parameters are computed by the event detector using the *Query Service* provided by the underlying RDF repository. After an event is detected, the event engine evaluates the *RDFTL Path expression* defined in the update operation. The result of the evaluation is stored in the system variable `$delta`.

In the centralized scenario, event parameters are easily accessed by rule components through the use of the system variable `$delta`. Regarding exchange of event parameters in distributed scenarios, the documentation does not mention how it is done. However, a different approach is needed as the event parameters are computed at the peer level and consumed at the super-peer level. As a consequence, communication by means of a system variable is not appropriate and a message-based communication

is required. A possible solution is to use an XML-based markup language to mark up the information and then, the event engine could send a message to the event handler with this information. Another solution could be to leave the computation of the affected fragments to the event handler. In this case, the event engine would still send a message with the event type and the event's occurrence time to the event handler (using the same approach as before). Then, the event handler could evaluate the triggered rule's event part in order to obtain the affected fragments.

Processing and Communication of Atomic Events

The prototype was built under the assumption that rules can only be triggered by events that occur locally, i.e. at a peer's single RDF repository. As a consequence no distributed event detection is possible. That is, an event occurring at a remote location can not trigger a local rule. This implies that there is no event communication or exchange among peers or super-peers. However, although not directly supported, distributed event detection is indirectly supported by the system. Following a rule propagation approach, rules are stored in all the relevant locations. That is, if a rule might be triggered by events occurring at remote locations, the system propagates and stores the rule at those locations; for this, each SP and peer keep information about distribution of rules across the network. Therefore, distributed event detection is indirectly supported in the system. Rules are still triggered by local events but, several copies of the rule exist in those places where an event may trigger the rule. This also implies that no exchange of events among peers or super-peers exist.

Although planned for future extensions, the support for distributed event detection and execution of actions opens up new issues. Among the most important ones we identify the following:

- *References to external fragments* If events occurring at one location may trigger rules stored at another locations, a mechanism for referring to fragments of metadata in remote RDF documents must be in place. A possibility here is to consider the expression `resource("uri")` of the RDFTL Path expressions when defining an event expression.

- *Remote detection of events* Clearly, a mechanism for detecting remote events must exist. Here, the most common approach is to implement an event engine that accepts event expressions form subscribers and then, when an event occurs, it notifies the interested subscribers by sending a message with the event parameters. Every peer in the network would have an event detector like this one. Notice that there is also the need for a method to locate the appropriate event detector. One possibility is to delegate this task to the ECA engine; based on the event expression the ECA engine would send the expression to the appropriate event engine in order to evaluate it.

- *Exchange of events* Related with the previous issue is the mechanism used for exchanging events among peers. The event detectors in the network must agree on a representation format for events. A possibility is to use an XML-based markup language, e.g. like with XChange messages [38] or ECA-ML in [7].

Integration and Reuse

Two problems hinder the possibility of integrating and reusing the current event detector prototype in other systems. First, the event detector is implemented as *wrapper* on top of an RDF repository, detecting events by monitoring the update operations. That is, the event engine is tightly coupled with the RDF repository. A solution could be to implement an event detector as a wrapper for RDF repositories in general. However, a second problem arise. Due to the fact that there is no standard update language for RDF yet, different repositories may support different update languages (and hence event languages). Thus, an event engine suitable for one event language (i.e. RDF repository) may not be appropriate for another. If a standard RDF update language is in place then, it could be possible to implement an event engine for RDF repositories and integrate it with an ECA engine. In this case, the ECA engine would control the execution of update operations and, after an update is executed, it could invoke the event engine with information about the operation just executed.

5.5.2 Active XQuery

Types of Events

The event language used by Active XQuery relies on the existence of an update language for XML. With this event language, only atomic events can be specified and detected in the system, i.e. there is no support for definition of composite events as no composers for combining atomic events are provided. Therefore, and as in the case of RDFTL, the expressivity of the event language clearly depends on the expressivity of the update language.

Furthermore, triggers are associated with updates operations executed on the XML repository at hand. As a consequence, the type of atomic events is closely related to the type of the update operations supported by the update language. The current formulation of the language allows the specification of events that reflect insertion, deletion, rename and replacement of fragments of XML, which are specified by XPath expressions.

As pointed out in [19], an important type of events that should be detected are the *schema-based* events. These events occur when update operations satisfy the constraints imposed by an XMLSchema, i.e when a constraint is satisfied. The current proposal does not consider this type of events.

Technique for Detecting Events

Events (modifications to XML data) are detected by the system when the rule engine executes directives. An *evalAfter* directives tells the rule engine to compute the set of `AFTER` triggers for a given expanded update. The statement given as parameter to the `COMPUTE_AFTER_ONFLICT_SET` function represents the update operation that have just changed the XML repository's state and thus, raised an event. This means that, when the rule engine executes an *evalAfter* directive it knows that the XML repository was affected by a data manipulation operation and so, it is time to take the appropriate actions. In the case of *evalBefore* directives, the difference is that such directives tell the rule engine to compute the set of `BEFORE` triggers. In this case, the parameter given to the `COMPUTE_BEFORE_ONFLICT_SET` function denotes the update operation that is about to change the repository's state, and thus about to rise an event. So, before executing the update operation, the rule engine must execute the corresponding `BEFORE` triggers.

An important characteristic of the technique used for detecting events is that there is no clear separation between the event detector and the ECA rule. There is actually no event engine responsible for monitoring the repository, computing event parameters and notifying occurred events. As such, directives play a central role in the detection of events. The approach to event detection proposed here is similar to the one where update operations are monitored by the system. Although the lack of an event detector as a separate module makes more difficult to extend and optimize its functionality, it seems quite natural to use this technique in this scenario. Specially when the system have to deal with `BEFORE` triggers, which must be triggered before the operations are actually executed. In this case, the system receives update operations and either immediately before or after their execution, an event is raised.

Event Parameters

In the system, the type of atomic events is implicitly determined by the type of expanded statement being considered (the type of update operation). This information is used by the rule engine when computing the set of triggered rules (conflict set for `AFTER` and `BEFORE` triggers). Based on the statement passed as parameter to the COMPUTE_BEFORE_CONFLICT_SET or COMPUTE_AFTER_CONFLICT_SET functions, the rule engine searches the rule base for the correspondent `AFTER` or `BEFORE` triggers whose event component matches the type of statement being considered.

The information about the XML fragments affected by an update operation is stored in the transition variables `OLD_NODE(S)` and `NEW_NODE(S)` [3]. These variables are instantiated during the processing step of each triggered rule; using information of the relevant fragments computed during the expansion process. Variables are accessed by using pointers, thus reducing the storage requirements. Regarding the time of occur-

[3] which variable to use depends on the trigger's granularity

rence of events, the rule engine does not actually compute it. However, if needed it can be set up after the directive is processed. In any case, event parameters are computed by the ECA rule engine.

The communication among rule components is by means of *transition variables*. These transition variables act as global variables that can be accessed by the functions implementing condition evaluation (condition part of rules) and action execution (action part of rules). Besides the already mentioned `OLD_NODE(S)` and `NEW_NODE(S)` variables, the language supports also the definition of transition variables at the rule level. That is, using *XQuery Let clauses*, it is possible to define variables whose scope is the trigger where they are defined; they can be referred to in the condition and action parts of triggers. As in the case of `OLD_NODE(S)` and `NEW_NODE(S)` variables, these variables are instantiated when each of the triggered rules is being processed, i.e. before evaluating the trigger's condition and executing the trigger's actions. In both cases, the rule engine is responsible for instantiating transition variables.

Processing and Communication of Atomic Events

The proposed system does not contemplate distributed event detection. Therefore, there is actually no exchange of events among systems. The reactive functionality is implemented on top of a single XML repository thus, events occur and are detected locally.

However, there is the possibility to specify an external operation in the action part of rules. External operations might be useful for sending messages to other systems about the occurrences of local events. Currently, the system does not support the execution of external operations and this is planned for future extensions. This possibility opens up a series of important issues that should be considered; among these we identify the following.

- *Specification of remote events* There must be a way of expressing interest in remote events, i.e. the event part of rules must allow the specification of events that occur at other locations. `BEFORE` triggers must be investigated very carefully.

- *Perception of remote events* If external operations executed as part of active rules in one node may inform other system about local events, there is the need for implementing a mechanism for receiving such notifications. Here, as with RDFTL we must define a format for exchanging event parameters and event instances among systems; XML-based markup language may help.

Integration and Reuse

Since the event detection functionality is tightly coupled with the ECA rule engine (it depends on the execution of directives), it is not possible to decouple it form the ECA engine. In the same sense, reuse of other event detectors is not possible.

5.5.3 XChange

Types of Events

Compared with other approaches, specially with those providing reactive functionality over XML or RDF repositories, the event language embedded in XChange is more expressive and general. The event language allows to model a considerable number of situations, ranging from low-level data manipulation in databases or XML/RDF repositories to high-level, application dependent situations.

Furthermore, the event language allows the specification of both atomic/primitive and composite events. Atomic events represent happenings in the Web. They can be the result of data manipulation operations executed on databases (local or remote) or messages received from other locations in the Web; they carry information about events that occur outside a system. On the other hand, composite events are defined in terms of atomic or composite events using the language's composers. These composers allow to specify temporal restrictions on a composite event's constituent events and temporal combinations of a sequence of atomic events.

Technique for Detecting Events

Event detection is based on evaluation of event queries against a stream of incoming atomic events. Each time an atomic event (its representation as event message) arrives, all the registered partial event queries are evaluated. Atomic event queries are evaluated by using a method called *simulation unification*. On the other hand, composite event queries are evaluated in an incremental manner, considering the tree-based representation of event queries. The event detector evaluates a composite event query by recursively traversing the event tree and evaluating the constituent event queries. Some important aspects of event query evaluation are the following:

- *Incremental evaluation* By using incremental evaluation, the event detector needs not to evaluate constituent event queries that were previously evaluated, thus reducing the computations needed to evaluate event queries (see example 5.3). Moreover, incremental evaluation allows to keep the evaluation cost per incoming atomic event constant. Incremental evaluation requires the answers to event queries to be stored in the system. In the prototype implementation, this is accomplished by storing event data (e.g. event parameters) at the nodes of the event tree that represents the event query being evaluated.

- *Event tree-based representation* The tree-based representation of event queries has some important properties. First, it allows to reuse or share common (sub)expressions within the same event query or between different event queries. For example, two or more event queries registered with the event detector may

use a common (sub)expression. Instead of creating separate event trees, an alternative is to construct an event graph that combines both event trees and stores the common (sub)expression only once. This alternative has been used in many event detectors (e.g. [19,27,56]).

Second, composite event expressions can be optimized by rewriting them into equivalent expressions using the language's operators properties. Notice that the techniques for rewriting event expressions vary depending on the event operators, i.e. techniques for SNOOP event expression may not be applicable to XChange event queries and vice versa. Optimization techniques help to reduce the computations and storage needs required for evaluating event queries.

Third, the tree-based representation is suitable for the implementation of several tasks related to event query processing. For example, the current prototype implements a deletion mechanism for deleting from event trees event instances whose time has expired.

- *Storage for event data* The event detection technique used in this prototype requires the use of suitable data structures for storing event data. For example, operator nodes must provide storage for the constituent events of the composite event they represent. In this case, the required data structures depend on the composer's semantics.

Event Parameters

For both atomic and composite events, event parameters are computed during event query evaluation. However, the techniques used for computing them differ from each other. In the case of atomic events, event parameters are computed during the evaluation of atomic event queries by the *simulation unification* method. As we said before, the result of evaluating an atomic event query contains the instantiation of the free variables appearing in the event query. These free variables constitute the event parameters. As for composite events, event parameters are computed based on the event parameters of their constituent events. In the current prototype this is done by invoking the constraint solver provided by Xcerpt [64]. Additionally, for both atomic and composite events, the event detector computes the `raising-time`, `reception-time`, `reception-id`, the `sender` and `recipient` of an event. For composite events the event detector also computes the `starting` and `ending-time`.

This approach to event parameter computation for composite events introduces modularity into the event detector architecture. The constraint solver can be replaced by a more efficient one, as long as a wrapper is in place.

Communication among rule components is by means of variable bindings. Variables instantiated during event detection are used for exchanging event-related attributes to other parts. By referring to these variables the condition and action parts

can have access to the event-related information (event data). Variables can be bound to event data (variables defined inside event queries), or to the result of an event query evaluation, i.e. an atomic or composite event.

Processing and Communication of Atomic Events

With XChange, events occurring in the Web are communicated between systems (XChange-aware systems) by sending event messages. They are communicated using a *push strategy*. This type of messages can be sent internally (posted) or between remote Web sites. Event messages are represented as XML documents. Incoming atomic events are processed by the event detector and serve as the basis for composite event detection. Moreover, clients (ECA engines or other systems like e.g. event detectors) can register atomic event queries with the event detector. In this way, the event detector can not only detect composite events but also atomic events. When atomic events are delivered to the event detector by e.g. event brokers, it then notifies the interested subscribers.

An important aspect to consider when discussing event communication is the method used to contact event detectors. That is, an ECA rule engine or event detector that needs to be notified about the occurrence of some event needs to contact another event engine. There might be many event detectors available in the Web, so an issue here is how to contact the right one. A possibility is to use a system similar to the DNS, where event detectors can be registered and then searched for by interested systems, e.g. ECA engines. Another possibility is that each event engine has a local list of trusted event detectors. A third possibility would be to use the same approach used in the discovery of Web Services.

The authentication issue must also be considered when dealing with registration and notification of events between systems. In the current prototype this is not considered. Due to the fact that this is a prototype implementation, atomic events are signaled to the system by “dummy" modules. Furthermore, the current implementation does not support communication through HTTP. Instead, only raw TCP/IP connections can be used for receiving and sending data. This is of course due to the fact that the current version is a prototype and the programming language’s support for Web protocols is poor.

An advantage of using XML to represent event messages is that it is possible to use XML query languages for extracting information from them.

Integration and Reuse

The current prototype implements the event engine as a thread (implemented by a recursive function[4]). The event engine communicates with the condition handler using

[4]actually every module in the system is implemented as a thread

a message-based communication channel and receives input data (atomic events) from a network link. All the information regarding registered rules and event trees is stored in the system and accessed by the event detector (and the other modules) through a parameter variable. As a consequence, the event engine is tightly coupled with the system itself, thus making impossible to reuse and integrate the current prototype in other systems. The selection of Haskell as the programming language has notably influenced the integration and reuse aspect of the event detector.

In order to solve this problem, what we need in principle is to include the information about event trees and event queries registered with the system inside the event detector. Events could be received and notified by using Web protocols for exchanging messages between the event detector and other systems. Here, a Web Service-based implementation would be an appropriate choice.

Chapter 6

Implementation of a Complex Event Detector

Contents

6.1 **Introduction** 80

6.2 **Architecture for Implementing the ECA Framework** 80

6.3 **Implementation Aspects** 82

6.4 **Evaluation of the Implemented System** 93

In this chapter we present the implementation of a prototype event detector for the general framework proposed in [7]. Its implementations is based on the results obtained from the previous comparative analysis of several event detectors. We have used a combination of techniques for processing incoming event instances, detecting events and representing event expressions. After describing its implementation we proceed to perform an evaluation of the implemented system. The evaluation is based on the criteria defined for the comparative framework and aims at detecting those aspects of the implementation that require further work. Part of the work described in this chapter was previously presented in [8].

6.1 Introduction

In this chapter, we present the prototype implementation of a complex event engine that detects events in the context of the *General Language for Reactivity and Evolution in the Web* [7]. Although this reactive language imposes no restriction on the type of event (sub)languages that can be used, we have focus our attention on a sublanguage of the event algebra SNOOP.

The design and implementation of such event detector was based on the results obtained from the analysis and comparison of different event engines presented in this work. We have used a combination of techniques for processing incoming event instances, detecting events and representing event expressions. However, due to the characteristics of the architecture proposed for the framework [7], the design and implementation of the event engine must consider important aspects such as the interaction with different language processors and the ECA rule engine, the evaluation of the so-called opaque expressions, the evaluation of composite expressions expressed in other event languages, the communication by means of variable bindings and the markup language for event expressions and variables.

Before focusing our attention on the implementation aspects and event detector itself, we give an overview of the architecture proposed for implementing the general ECA framework.

6.2 Architecture for Implementing the ECA Framework

The implementation of the ECA framework proposed in [7] requires an architecture that, not only considers the distributed aspect of ECA rules but also the heterogeneity of languages and concepts. Recall from chapter 4 (section 4.4) that the specification of ECA rules in the framework requires the use of appropriate languages for each component. The event, condition and action parts of rules are specified by expressions of an event, condition (query), and action language respectively. Furthermore, the ECA framework allows the flexible combination of these languages, thus it is possible to define rules of a rule base using different event, conditions and actions languages. When defining ECA rules, every rule component is associated with the language used for its specification by using a language identifier, like for example the language's URI.

The architecture proposed in [7, 54] is a service-based, distributed architecture where every language is associated with a Web Service that implements the language. For example, for an event language, a service would implement the semantics of the language's composers and the processing of atomic events. A Web Service for a language used for specifying the condition part of rules would probably implement a query language, whereas an action language would be implemented by a service capable of executing local and distributed actions. A valid assumption here is that these

Web Services may be available at the URI specified when defining ECA rule components. For example, the Web Service implementing an event detector for a given event algebra may be accessed at the event algebra's URI.

This service-based approach enables the architecture to accommodate a good number of Web Services required for executing ECA rules in the Web. Among these we may have *ECA rule engines*, *atomic and composite event detectors for application-dependent events*, *event detectors for application-independent events*, *Web transactions engines*, *query processors*, *underlying active databases* and *action processors*. Notice that, in order for these components (Web Services) to be integrated in the architecture they must support the communication with other components by means of variable bindings. Figure 6.1 illustrates the proposed architecture. Each composite event detector implements an event algebra and thus, it detects composite events defined by expressions of that algebra. For example, we may have one event detector implementing the SNOOP language and another for the event language defined in XChange. ECA rule engines may query and update Web sites as part of the rule execution mechanism. Atomic event detectors are responsible for detecting atomic events in the Web. The may provide atomic events directly to the composite event engines or to event brokers. An event broker acts as an intermediary between atomic and composite event detectors. We may have for example, event brokers associated with certain type of Web sites. The interaction mechanism between atomic event detectors and composite event detectors can be *application-centered* or *language-centered* [7].

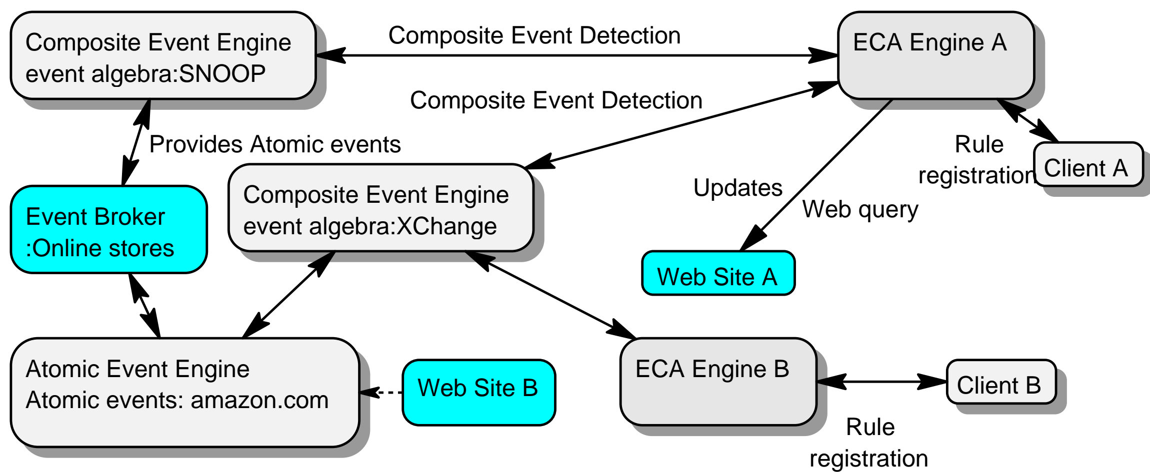

Figure 6.1: An architecture supporting the proposed ECA Framework

ECA rules can be processed locally at the nodes where they were defined and stored or, on the other hand, they can be registered at an *ECA rule engine* (a Web Service implementing a rule engine) that processes registered rules. In order to execute a rule, an ECA engine uses the URI references specified in each of the rule components to invoke the Web Service providing the language processor. Notice that in this scenario, several event detectors, query engines and action processors may be available as Web Services in the Web. Then, depending on the URI associated with each component

(language's URI), the ECA engine interacts with them. Furthermore, an event engine may need to interact with several other event evaluators as the event expressions registered with it may use different (sub)languages to define (sub)expressions. Then, the event detector needs to contact the appropriate Web Services in order to evaluate these (sub)expressions. The same is valid for query engines and action processors.

6.3 Implementation Aspects

This section is devoted to the discussion of the most important aspects of the prototype implementation. We will see in some level of details how the event detector was implemented. In particular, we will describe the architecture of the event detector, how it handles event expressions and the mechanism used for detecting composite events. This information is complemented with the class diagram presented in appendix A.

6.3.1 Architecture of the Event Detector

The event engine was developed in the context of the general ECA framework presented before. Therefore, the it assumes events are registered by a general ECA engine and marked up using the XML-based markup language for ECA rules proposed for the framework. The prototype was fully implemented in Java as a Web Service and uses the API provided by [9] [1]. This Java library implements an evaluation engine according to the framework introduced before and it provides the set of classes implementing expressions in our language as well as the web services functionality.

The event engine detects composite events specified by event expressions of an illustrative sublanguage of the SNOOP algebra [28, 55]. Clients (ECA engines or other components in the architecture) register event expressions with the event engine when they want to be notified of the occurrence of events. Along with the expressions, clients define the variables (input and output) associated with the expressions they register. All the data necessary to detect an event is passed to the engine by using input variables (variable-value pairs). Output variables are used by the event engine in order to communicate the result of an event occurrence. Therefore, the communication mechanism between the engine and its clients is implemented by means of variable bindings. Additionally, the result of an event evaluation comprises the set of event instances that contribute to the detection of the event. For example, consider the atomic event defined by the expression `newBook(BookTitle,"Dan Brown",BookPrice)`; upon detection of availability of a book whose author is "Dan Brown", the engine notifies the client and sends it the bindings for the variables specifying the title of the book and its price. Notice that some of the variables may be bound from the start (input variables), e.g.

[1]Note: the event detector was implemented using the version of the R3 prototype (API) available at the moment of writing this thesis. Current versions of the API may provide extra functionality not implemented in the event detector

like the implicit variable `Author` in the example above whose value expresses interest in books of the given author. In the same way that event expressions are marked up, the result of an event evaluation is also marked up using an XML-based markup language.

The event engine assumes that the constituent events of a composite event are detected outside the system (by atomic event evaluators); after atomic events are detected, they are signaled (provided) to the event engine. Incoming atomic events are processed by the event engine and combined according to the semantics of the language's operators used to define the composite events. In the current prototype implementation, the event engine uses a dummy atomic event evaluator that provides atomic event instances to the engine. The result of evaluating atomic events is again a set of variable bindings and the atomic event instance that was detected.

The event detector's architecture comprises three main modules: an *expression handler*, an *event graph handler* and an *atomic event instances handler*.

- The expression handler is responsible for accepting event expressions and checking that expressions are syntactically correct. It accepts registration requests from clients through the event detector's interface and once the expression has been registered it passes the control to the event graph handler.

- The event graph handler is responsible for representing and storing event expressions using event trees. Event trees are combined into an event graph in order to exploit commonalities among expressions and reduce the memory requirements. This module is also responsible for processing the incoming atomic event instances signaled to the event engine and detecting the composite events for which these instances are relevant. In summary, this is the most important module of the event detector as it implements the composite event detection. This module is implemented by the class *EventGraph*.

- The atomic instances handler is in charge of pre-processing the incoming atomic event instances detected outside the system and signaled to it by means of messages. For every new event instance that arrives to the system, this module checks whether the instance contributes to any of the atomic event expressions currently registered with the system. If the instance may be used for at least one atomic expression then this module passes it to the event graph module for event detection, otherwise it is discarded.

The external actors of the system include *clients* that want to be notified about the occurrence of composite events and *atomic event brokers* which are responsible for providing (constituent) event instances to the event detector. We may have several of these event brokers, for example one for every different domain from which the event detector can receive constituent event instances.

Figure 6.2 depicts the event engine's architecture as well as the relationship between event brokers, clients and the event detector.

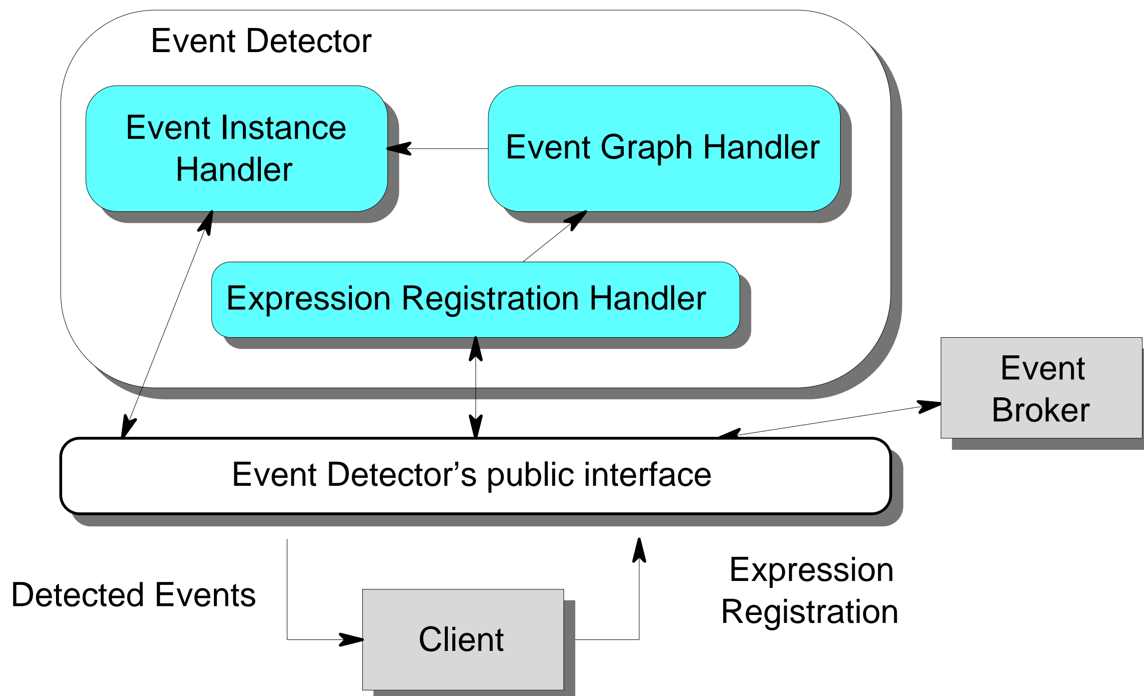

Figure 6.2: The architecture of the event detector

The event detector is implemented by the class `EventDetectionEngine`, which is a subclass of the abstract class `Evaluator` provided by the R3 API mentioned before. The event detector runs as a Web service using the SOAP implementation Apache Axis [2]. For additional information about the R3 Java library the reader is referred to [9].

6.3.2 Event Model and Event Expressions

In the system, an event is a happening that occurs somewhere in the web, i.e. at some location and at some point in time. An event can be the insertion of a tuple in a database or the update of an XML or RDF repository. Also, events can be high-level application-dependent happenings such as the *cancellation of a reservation on a flight from Lisbon to Chicago on September 17*. Events are classified as *atomic events* or *composite events*. Atomic events represent happenings outside the system and are notified (in a push manner) to the system by messages. From the event detector's point of view, an atomic event is something that happens in the web and is received along with the event's information. For example, an atomic event may reflect the availability in stock of a movie DVD (in the Web of movie and music selling). Such atomic event may be specified by the atomic event expression `newMovieDVD(MovieTitle,DVDPrice)`. Composite events, on the other hand, are complex events specified by event expressions written in some event algebra and defined in terms of atomic and com-

[2] `http://ws.apache.org/axis/`

posite events. They are detected by the event engine. For example, the event `newCD(CDTitle,CDArtist)OR newBook(BookTitle,"Dan Brown")` is a composite event. Notice that, atomic events in the system may be composite events in another systems.

The event algebra implemented by the event engine is an illustrative subset of the SNOOP event language. This sublanguage includes the disjunctive operator *OR*, the sequence operator *SEQ* (also denoted as ';') and the conjunctive operator *ANY*. For a formal specification of the semantics, the reader is referred to [27]. Expressions of this event language denote the events that the event engine must detect. The engine accepts and processes three types of event expressions: *atomic*, *composite* and *opaque* expressions. *Atomic* and *composite expressions* specify atomic and composite events respectively, as discussed above. Composite expressions use the language's operators to define complex events based on atomic events and previously defined composite events. On the other hand, *opaque expressions* denote events that occur outside the system, i.e. they are treated as atomic expressions; and they can not be further decomposed into more basic events. The difference here is that opaque expressions define events using some external language (understood by some other language engine). For example, an opaque expression may define an event using Java code, Prolog code, SQL code or any other language, provided that an engine for evaluating such language exist.

6.3.3 Markup Language for ECA Rules

Event expressions and associated variables are represented using variant of the XML-based markup language proposed in [7]. In this markup language, event expressions are defined by specifying the event, condition and action parts, as well as the variables used for communicating relevant information. In accordance with the proposed framework, each rule part defines the URI associated with the language used for specifying that part. The XML elements used for marking up expressions depend on the type of expression being represented.

- *Atomic expressions.* Atomic expressions are represented by specifying the construct's name (using the *operator* attribute), the variables declaration (*with* element), the construct's parameters (using *having* and *parameter* elements) and, the relationship between variables and parameters. In Figure 6.3 on the next page, lines 4 to 9 show the definition of the atomic event expression `newMovieDVD(MovieTitle,DVDPrice)` [3].

- *Opaque expressions.* Opaque expressions are represented by specifying the language's URI, the opaque content to be evaluated and the variables declaration. In Figure 6.3 on the following page, lines 14 to 18 show the definition of an opaque expression.

[3] namespace declaration is omitted for simplicity

- *Composite expressions.* Composite expressions are represented by specifying the operator's name, the language's URI, the arguments (sub expressions) to which the operator is applied, the variables declaration (input and output variables) and, the operator's parameters if needed. An example of a composite expression built out from atomic and opaque expressions is depicted in Figure 6.3.

```
 1 <eca:expression operator="OR" language="http://snoop-r.org">
 2  <eca:argument name="left">
 3   <eca:solve>
 4    <eca:expression operator="newMovieDVD" language="http://movies.com">
 5     <with><Variable name="MovieTitle" mode="bind"/></with>
 6     <with><Variable name="DVDPrice" mode="bind"/></with>
 7     <having><parameter name="mtitle" bindVar="MovieTitle"/></having>
 8     <having><parameter name="mprice" bindVar="DVDPrice"/></having>
 9    </eca:expression>
10   </eca:solve>
11  </eca:argument>
12  <eca:argument name="right">
13   <eca:solve>
14    <eca:expression language="http://languages.org/prolog">
15     <literal>
16     ?- newCD(ListOfCDs), member(cd(myCD),ListOfCDs).
17     </literal>
18    </eca:expression>
19   </eca:solve>
20  </eca:argument>
21 </eca:expression>
```

Figure 6.3: A composite event expression.

Variables are represented by specifying their name, their value and their mode (used for defining input and output variables). Input variables are defined by setting the attribute mode's value to *use* and in this case, the variable's value must be specified (using the *literal* element). Output variables are defined by setting the mode to *bind*. Additionally, variables are related to the parameters of atomic events using the attribute *bindVar* in the *parameter* element. Line 5 in Figure 6.3 shows the definition of variable *MovieTitle*.

6.3.4 Representing Event Expressions: Event Trees and Event Graph

Event expressions are stored in the system by using event trees. Leaf nodes implement atomic and opaque expressions (atomic event types), while inner nodes implement the language's composers (composite events). Since different event expressions registered with the event engine may use the same subexpressions and, different clients may register the same expression, the detector uses an event graph in order to combine

different event trees and support the reuse of common (sub)expressions. That is, if two event expressions *A* and *B* use the same subexpression *C*, the event tree representing the expression *C* is shared by the event trees representing the expressions *A* and *B*. This means that the event graph will store the information associated with the event tree being reused only once, as opposed to creating the same (sub)tree more than once. As a result of this combination, every leaf in the event graph may have several parents. The event graph is thus a collection of event trees.

Example 6.1 ***Sharing event trees.*** *Consider the following two composite expressions where* newBook, newCd *and* newMovieDVD *represent atomic event types:*

```
A = OR(newBook(BookTitle,"Dan Brown"),newCD(CDTitle1,"U2"))
B = SEQ(newCD(CDTitle2,"The Corrs"),newMovieDVD(MovieTitle,"2006"))
```

Both expressions share the subexpression `C = newCD`*, i.e. the atomic event type* newCD*. As a consequence, the event trees representing the expressions* **A** *and* **B** *will share the (sub)tree associated with* **C** *(a leaf node in this case). Figure 6.4 illustrates the event trees and the resulting event graph.*

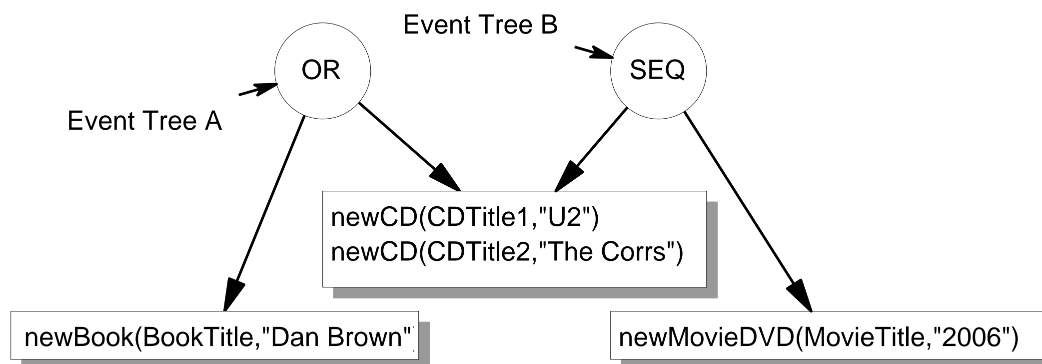

Figure 6.4: An Event Graph sharing two Event Trees

In the system, the event graph functionality is implemented by the class `EventGraph` whereas event trees are implemented by the class `EventTree`. Class `EventGraph` provides methods for storing an event expression in the event graph, deleting event expressions and processing incoming event instances signaled to the system and provided to the event graph by the atomic instance handler. The R3 library provides the implementation for event expressions.

6.3.5 Registration of Event Expressions

When a client registers an event expression, the event engine tries to construct the event tree that represents the expression. The construction of event trees is an incremental, bottom-up procedure that starts at the leaf nodes representing the atomic event types

defined in the expression and ends at the root node (an operator node). Once this process reaches the tree's root, the event engine adds the event tree to the event graph and the registration process ends. During this process, syntactically incorrect expressions are discarded and their incomplete event trees are deleted.

- *Processing of atomic expressions* When an atomic expression is processed, the detector checks (based on the operator's name) whether the leaf node representing the atomic event type described by the expression is already in the graph. The event engine does this by accessing a hash table that contains all the atomic event expressions registered with the system. If that is the case, the event engine reuses the node, otherwise a new leaf node is created and added to the event tree being constructed. Notice that the event engine needs to distinguish between an atomic event type (its definition) and the events (instances) of that event type. In the system, every leaf node stores information about the events it represents. This information includes the variables to be bound as a result of the event occurrence (output variables), the rule or expression the event belongs to, the parent to be activated when the event occurs and an evaluation's ID (implemented as an URI) that uniquely identifies the evaluation of the expression. Every leaf may store event instances from different expressions (all of them of the same event type) as nodes can be shared.

- *Processing of composite expressions* The processing of a composite expression depends on the class of composite expression being considered. Here we distinguish two cases. In the first case, if the composite expression is part of the event algebra being implemented, the event detector decomposes it into smaller subexpressions and processes them recursively. In the other case, if the expression belongs to a different language (an external expression), the event engine treats the event denoted by the expression as if it were an atomic event and so, it creates a leaf node to represent the event. From the detector's point of view, external expressions are treated as expressions denoting atomic events; i.e the event engine will issue a call to another evaluator to evaluate the expression and then process the results (the variable bindings).

- *Processing of opaque expressions* When the event detector processes an opaque expression, it creates a new leaf node, stores information about the variables used in the expression (input and output variables) and then adds the event tree (a single node) to the event graph. In this case there is no reuse of event tree nodes.

6.3.6 Composite Event Detection

The detection of composite events follows a bottom-up process that starts when an atomic event instance is signaled to the system. Incoming atomic events are processed

according to the recent context defined in [55]. When the event detector receives a message indicating that an event instance e_i of an atomic event type E_i has occurred, it computes the instance's occurrence time, stores the instance's information (variable bindings) in the leaf node associated with E_i and then activates the node. Event instances are propagated from the leaves up to the event tree's root. Events are detected when the processing reaches the event tree's root. The detection of composite events involves several aspects.

Evaluation of Atomic Expressions

Events denoted by atomic expressions, opaque expressions and external composite expressions are detected outside the system. Therefore, the event engine evaluates these expressions by invoking another event evaluator (event provider or broker). In order to evaluate these expressions, the detector sends to the event provider the expression to be evaluated together with the variable bindings for the input variables used in the expression. As a result of this operation, the event detector receives either an evaluation error or an evaluation ID (again an URI) that uniquely identifies the expression's evaluation. If the result of such evaluation is a not-null evaluation ID, this means that the event provider will evaluate the expression asynchronously and then communicate the results. In this case, what the event engine must do is to store this ID in order to process future results. This ID is stored at the leaf representing the event type denoted by the expression that was just evaluated. On the other hand, if the detector receives an evaluation error, it marks the event expression as failed and starts a process to check whether the composite expression is active or not (it still can produce events).

Event expressions evaluated outside the system require the event engine to be able to contact a valid event evaluator (broker) for the given expression. In a realistic scenario this would require the use of Web services providing event detection capabilities for the types of event expressions being evaluated outside the system. At the moment, the current prototype implementation implements this by a "dummy" module that provides the required functionality.

Activating Leaf Nodes

When a leaf node is activated, the event engine validates the event instance stored at the node and then, if the validation is successful, it propagates the instance to the appropriate parent and activate it. If an event instance is not valid, the event detector discards the instance. An event instance is valid if the expected output variables were instantiated during the evaluation of the instance's event expression and, the result of this evaluation contains no extra output variables. In other words, an event instance is valid if the variable bindings produced as the result of the event evaluation are valid.

In order to validate a set of variable bindings we employ the following technique: Let's assume that after an event instance is notified to the system the variable bindings

are available as a list in the variable `Tuple`. Let's also assume that the output variables of an atomic event expression are available as a list in the variable `FreeVars` and that the input variables associated with a composite expression are stored in the variable `Input`. Then, the variable bindings produced by an atomic event provider are valid if every variable V_i in `Tuple` is either defined in `FreeVars` or in `Input`. Additionally, every variable V_j in `FreeVars` must be defined in `Tuple`. For example, suppose that the event detector is waiting for the atomic event `newBook(Title,"Dan Brown",price)` and `Title` is an output variable in `FreeVars`. If the event instance received by the detector is `newBook("The Da Vinci Code","Dan Brown","23.90")` but no variable bindings are produced, the event engine cannot bind the variable `Title` and hence the event instance must be discarded. In the same way if the variable bindings contain something like `Tuple = [<Price,"23.90">]`, the event detector must discard the instance as the variable `Title` is not bound and it should be.

Activating Operator Nodes

Although the computation performed by an operator node depends on the operator's semantics, every operator performs a series of common tasks upon activation. These tasks include storing the event instance propagated from its children, computing the variable bindings associated with the composite event represented by the node and notifying composite events if needed.

Computing Variable Bindings

When an operator node is activated, the event detector computes the variable bindings associated with the composite event being detected. That is, variable bindings from all constituents events are joined; the constraints among variables must be satisfied. For example, in the expression `SEQ(newCD(T,"U2"),newMusicDVD(T,"U2"))`, the values of the variable `T` must coincide. That is, the output variable of atomic event `newCD` is an input variable for the event `newMusicDVD`. Notice that for operators `SEQ` and `ANY` the variable bindings are joined but, for operator `OR` this is not necessary; the variable bindings at the operator `OR` are the ones provided by the event instance that has occurred.

Notifying Composite Event Occurrences

When a composite event is detected, the system must return the variable bindings associated with the composite event and the sequence of events that have contributed to its detection[4]. Additionally, the event engine informs the client whether the composite event expression will produce future results or not. To notify this, the event engine returns the expression's next evaluation ID that uniquely identifies the next evaluation of

[4]the sequence of constituent event instances is marked up using an XML-based markup similar to the ECA-ML

the expression. In our case this value is always the expression's ID that was generated at the expression's registration time. So, if there is a future evaluation, the expression's next evaluation ID is set to the expression's ID. But, if the event cannot occur anymore (e.g. because one of its constituent events will not occur again), this value is set to `null`.

Example 6.2 ***Notification of detected events.*** *Suppose the event expression* `A = OR(newCD(CDTitle,"U2"),newBook(BTitle,"Dan Brown"))` *is registered with the event engine with* $ID = 1$[5] *and that at time* t_1 *the event instance* newCD("Boy","U2") *occurs; with variable bindings* `Tuple=[<CDTitle,"Boy">]` *and events sequence* `Literal=<expression operator="newCD"/>`*. As a result of this, the event engine detects the composite event* `OR` *at time* t_1 *and returns a result where:*

```
NewEvaluationID = 1
Variable Bindings = [(CDTitle,"Boy")]
Sequence of events = "
 <expression operator="OR">
  <argument>
   <expression operator="newCD"/>
  </argument>
 </expression>"
```

Implementing Operator OR's Semantics

When the node is activated, the event instance propagated from one of its children is stored at the node and a composite event instance is created; the propagated instance is its only constituent event. This composite instance is then propagated to the node's parents.

Implementing Operator SEQ's Semantics

Upon activation, the `SEQ` operator stores the propagated event instance. Since the operator might be shared by different expressions, the event detector keeps separate storage for instances of different expressions. Moreover, since the order of the operator's constituent events matters, the event detector stores left and right children separately. When the operator node is activated, the event engine checks which child has occurred. If the right child has occurred, it checks whether the left child has occurred. If so, it compares the events' occurrence time and; if the time constraints are satisfied, it computes the variable bindings. If the left child has not occurred yet, the right child is discarded.

[5] although the ID is an URI we use numbers for simplicity

Implementing Operator ANY's Semantics

The storage requirements for this node are similar to the `SEQ` operator's. The only difference is that the order of the children does not matter. When the operator node is activated, the detector stores the event instance and then, it checks whether the number of constituent events already detected is equals to the operator's parameter (*numberToDetect*). If so, the time restrictions are checked; if they are satisfied, the variable bindings are computed. After this, the event detector constructs a composite event instance and activates the appropriate parent. If, on the other hand, the number of detected events is less than the number of required events, the event detector continues normally.

6.3.7 Deletion of Registered Event Expressions

Event expressions registered with the system can also be unregistered. An expression is deleted from the system by deleting from the event graph the event tree representing the expression (expressions are identified by a URI associated with them). Since an event tree may reuse leaves and subtrees of another event trees, the detector checks whether a node in the event tree is being used by another tree before deleting it. In order to delete an event tree, the detector visits it in pre-order mode and processes every node according to its type. If the node is a leaf and is not being used by any other event tree, the node is deleted from the event graph; otherwise it is left. In any case, the event engine deletes the expression's related information from the leaf, i.e the event expression stored at it. Now, if the node is an operator node, the event detector checks whether the node is being used by another event tree. If the node is not being used, the detector deletes the event trees associated with the node's children and then deletes the operator node from the graph. If the node is used by other trees, the event detector deletes the event trees associated with the node's children as well as any composite event instance stored at the node but, it keeps the node in the graph. Let's illustrate this with an example.

Example 6.3 ***Deletion of Registered Event Expressions.*** *Let's suppose that at a given point in time the following two event expressions are registered with the event detector:*

```
A = OR(newBook(BT,"Dan Brown"),newCD(CDT1,"U2"))
B = SEQ(newCD(CDT2,"The Corrs"),newMovieDVD(MT,"2006"))
```

Figure 6.5 on the facing page depicts the event graph before and after eliminating the event expression B.

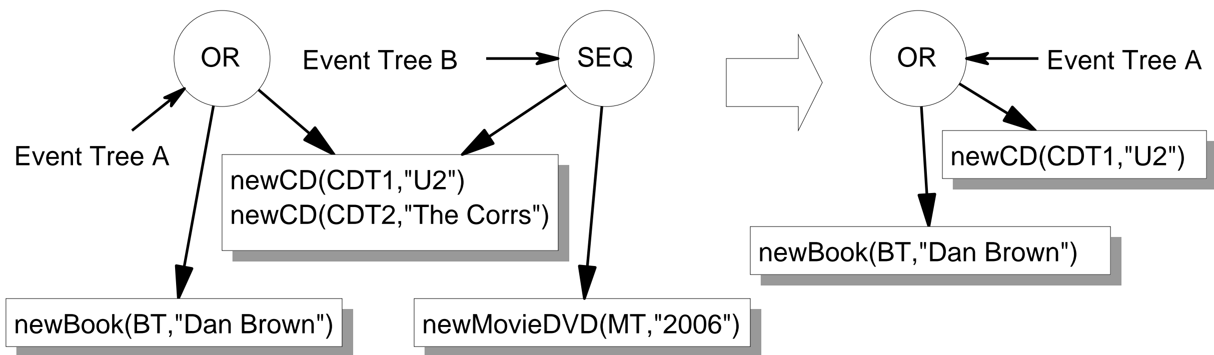

Figure 6.5: Event graph before and after deleting expression B

6.4 Evaluation of the Implemented System

In this section we evaluate the implemented event detector according to the criteria previously defined for the comparative framework. The aim of this evaluation is to highlight the limitations of the current implementation and detect those aspects of the prototype that require improvements and extensions.

Types of Events

The prototype developed for this active language implements a generic event language; a sublanguage of the event algebra SNOOP. As such, it allows the specification of both atomic and composite events. The later ones are defined by combining atomic and composite events with the language's operators. Furthermore, and in contrast with other approaches, the event language allows the definition of composite event expressions whose sub expressions are written in another event language or programming language (using the so-called *opaque expressions*). Event expressions in the language can be used for modelling any type of situations, ranging from low-level data manipulation in databases or XML and RDF repositories to high-level, application dependent situations. Moreover, atomic events may occur remotely at other Web sites.

Although the event language implemented by the event engine is expressive enough to analyze several aspects of event detection and, it allows to model a reasonable number of situations, it must be extended to provide a more expressive set of operators. This will allow the modelling of a wider range of situations.

Technique for Detecting Events

The implemented prototype detects composite events expressed by event expressions registered with the event detector. The event detector assumes that event expressions are previously registered with some ECA engine as part of some ECA rule; however, this does not represent a restriction as event expressions could be, in principle, registered by any entity requiring detection of events.

The event engine represents event expressions using a tree-based approach. Leaf nodes represent atomic event types defined by atomic event expressions or opaque expressions. Inner nodes (operator nodes) represent composite events and implement the operators' semantics. Moreover, nodes provide storage for event data, i.e. event instances detected so far and event parameters (variable bindings). Composite events are detected when atomic events are received and combined according to the operators' semantics; an incremental evaluation of event expressions as with XChange. Incoming atomic events are injected at the leaf nodes and from there they are propagated in a bottom-up fashion up to the event tree's root. Furthermore, the event engine relies on the existence of language processors (event detectors in this context) for detecting composite events. As composite event expressions may contain (sub)expressions from other languages, the event detector needs to contact other event engines so as to evaluate those (sub)expressions.

Our decision to use event trees was influenced by the advantages this approach provides. One of this advantages is that it is possible to exploit commonalities among event expressions. In particular, the event engine combines event trees into an event graph thus, reusing common (sub)expressions (see example 6.1). As a consequence, storage requirements can be reduced as same expressions are not stored more than once. Moreover, a single atomic event may contribute to detect different composite events, thus reducing the computations needed in order to detect them. Two important issues arise when expressions can be shared using event trees.

- *Deletion of event expressions* Event expressions registered with the system may be unregistered by the subscribers. This implies that the event tree representing the expressions must be deleted form the event graph. Here, care must be taken to avoid eliminating a node that is being used by other event trees. In this prototype, the event engine traverses the correspondent event tree and deletes from it those nodes that are not being used by other trees. This also shows that the tree-based representation is suitable for the implementation of different tasks regarding event expressions.

- *Registering expressions* Imagine a situation where the event expression `A = E1 AND E2` is registered with the event detector and the leaf node representing `E2` contains an event instance at time t_1. Let us suppose that event expression `B = E2 OR E3` is then registered with the event engine at time t_2 and that, as a result of expression optimization the leaf node for `E2` is shared among event trees. A question here is whether or not the event instance stored at `E2` should be considered for expression `B`. The approach taken in this prototype is to ignored any previous event instances stored at the nodes of an event tree. Only event instances occurring after the expression registration time are considered. An alternative would be to built a different event tree for every expression but, this would cause to lose the

advantages gained from expression optimization. Another possibility is to filter out event instances immediately after an event expression is registered.

The event engine implemented by this prototype could be extended and improved in several ways, specially by reusing and adapting previous techniques used for event detection. In this respect, we identify some key aspects that must be revised and/or extended.

- *Type system for variables* The current prototype does not take into account the type of variables. Using types we can implement additional checks when detecting events. Here, as proposed in [7], an ontology specifying a type system may be used as it allows for sharing of values.

- *Computation of variable bindings* The approach used in XChange [38] for computing variable bindings seems to be adequate for this event detector. In principle, it would be possible to invoke a constraint solver in order to compute the variable bindings. However, a deep analysis of the performance of this and other alternatives should be made.

- *Expressive power* Although the language implemented by the current prototype is suitable for a wide range of applications and is expressive enough to analyze different aspects of event detection, we would like to extend its expressivity. In this sense, the implementation of cumulative operators (SNOOP language) would provide a higher expressivity, extending in this way the applicability of the evaluator. Also, other parameter contexts could be implemented.

- *Detection window* A *detection window* specifies the period of time during which event instances stored in the system are still valid, i.e. they can contribute to the detection of composite events. The use of a detection window ensures that event instances are eventually discarded from the event engine, reducing in that way the storage requirements. The current prototype does not implement a detection window. That is, event instances are stored in the event graph until either a composite event is detected (and so instances that contributed to the detection are deleted) or the corresponding event expression is unregistered. However, if we consider that the event engine may receive a considerable number of event instances along its history, keeping instances in the system forever is not a good option. To cope with this problem several alternatives exist. For example, in [52] every rule is associated with a time interval. This interval determines the size of the detection window for every particular rule. Periodically, a function checks the event instances stored in the system for every rule. Those instances that do not fall inside the detection window are discarded. The advantage of this approach is that we can specify a different detection window for every event expression, e.g. based on an estimate of the frequency of atomic event occurrences for the

type of events specified by the expression. However, this technique requires to select an appropriate time interval for performing periodical checks.

- *Optimization of event expressions* As pointed out in [28], an event can be specified by several equivalent expressions. In this sense, rewriting techniques may simplify the detection process and uncover common (sub)expressions. For example, the event expression `(E1 ; E2)OR(E2 ; E1)` can be transformed into the equivalent expression `E1 AND E2`, which is simpler to process. Optimization of expressions is an important area of future works as the event detector is expected to receive and process an important number of event expressions. However, this would depend on the final set of event operators implemented by the system.

- *Dealing with delays* In a distributed environment such as the Web, it is common to experiment delays in the network communications. An event detector receiving atomic events from remote systems should consider the fact that event instances may suffer from such delays. Delays may cause out-of-order event delivery and thus, incorrect event detection. Therefore, a strategy for dealing with late notifications must be implemented. In [52], the authors show that a simple alternative is to order the events in a *global ordered event history* based on the maximum delay time and the current time. However, as they mention in their work, this strategy have important disadvantages. Instead, they propose an alternative solution that deals with delays during the event detection process; this eliminates the need for an ordering phase before the event detection phase. The strategy is based on a *hierarchical event history* (stored in the event tree) and *scheduled time events* that specify the tolerated delay time for events. For details of the strategy the reader is referred to [52].

Event Parameters

In the case of atomic events, event parameters are computed when the events are detected outside the system. Instead, event parameters for composite events are computed by combining the event parameters of the constituent events. In both cases, event parameters are represented as variable bindings, i.e. *(variable, value)* pairs. In order to communicate event parameters among rule components, the variable bindings produced as a result of an event detection are marked-up using the XML-based markup language proposed for the general framework. Regarding event parameters two aspects require further considerations and improvements.

- *Type system for variables* The current prototype does not take into account the type of variables. Using types we can implement additional checks when detecting events. For example, when the atomic event `newBook(BookTitle,BookPrice)` is signaled, the event engine can check whether or not the value assigned to the variable `BookPrice` is a float number or a string. If the value represent a string

instead of a float number the event instance may be discarded. Here, as proposed in [7], an ontology specifying a type system may be used as it allows for sharing of values.

- *Computation of variable bindings* The approach used in XChange for computing variable bindings seems to be adequate for this event detector. In principle, it would be possible to invoke a constraint solver in order to compute the variable bindings.

Processing and Communication of Atomic Events

A key aspect that must be revised in the current implementation is the way the event engine locates and invokes other event evaluators. Currently, atomic event detection, evaluation of opaque expressions and evaluation of expressions written in other event algebras is implemented by "dummy" modules. However, event detectors for the Web will most likely be implemented as Web Services. This in turn calls for a mechanism for locating appropriate event engines (there might be many of them) and ultimately requesting the evaluation of a given event expression. This can be solved by maintaining a list of trusted services in the event engine or by requesting this information to the ECA engine. Alternatively, the interaction between the event engine and other event evaluators (language processors) may follow a *language-centered approach* [7]. Although not fully implemented, the event engine identifies event detectors or language processors by using the language's URI specified in each (sub)expression.

Communication of atomic events follows a *push-strategy*. That is, atomic events are detected outside the system and then signaled to it. This helps to reduce the network traffic as events are not broadcasted to every system in the network. Moreover, it helps to improve the efficiency of the event detector as atomic events are received as soon as they occur; assuming reasonable network delays.

Integration and Reuse

In order for the event engine to be able to interact with other systems (e.g. ECA rule engines or event detectors), the only requirement is to implement the communication by means of variable bindings.

An aspect that should be considered and implemented in the event engine is an authentication method for registering event expressions. If the event detector receives all the event expressions from an ECA engine, then this mechanism may be already implemented in the ECA rule engine. However, if the event engine can receive evaluation requests from other systems then, the authentication system may be implemented inside the event detector.

Chapter 7

Conclusion

Event detection constitutes a key aspect of the design and implementation of different computational systems and more specifically, *reactive systems*. This topic has been extensively studied in different research areas including *Active Databases*, *Event-driven Architectures*, *Distributed Systems*, *Concurrent Systems* and more recently, the *World-Wide Web*.

In this work, we have studied event detection in the context of the Web. More specifically, our goal was to develop a complex event detector for the Web that could be integrated into the ECA framework proposed in [7]. However, in order to achieve our goal we have first analyzed and compared the use and implementation of event detectors in the context of active languages for the Web. In summary, the contributions of this work are the following:

- *Comparative analysis.* As a first contribution, we have defined a comparative framework to analyze and compare different active languages for the Web and in particular, to study the way these languages employ event detectors. We have focused our attention on the active languages: RDFTL (RDF Triggering Language), Active XQuery and XChange.

  The aim of this analysis was to study different implementations of event detectors in order to learn about alternative approaches to event detection both in Active Databases as well as in the Web context. More specifically, the analysis was aimed at producing a series of a guidelines for the implementation of such systems. In this sense, we wanted to formulate guidelines that capture the key aspects of these systems in order to ease the implementation of this type of applications.

  The analysis was based on a serie of criteria that we considered are important when evaluating event detectors. More specifically we have compared the event detectors in terms of *the technique used for detecting events*, *the types of events that can be detected*, *the processing of event parameters*, *the techniques used for communicating atomic events* and *the possibilities to reuse and integrate them in existing systems*.

- *Implementation of an Event Detector.* Using the results obtained from the compar-

ative analysis, we have implemented a prototype of an event detector for the Web, which can be integrated into the ECA Framework defined in [7]. The event detector detects composite events expressed by expressions of an illustrative sublanguage of the event algebra SNOOP. A novel feature of this event engine is its ability to detect events expressed by *opaque expressions* [7]. This in turn, extends the event detector's applicability as events can be expressed using not only event languages but also other languages such as Java and Prolog. The system detects composite events by processing and combining atomic events that occur in the Web. The communication with other services is implemented using variable bindings.

7.1 Design and Implementation of Event Detectors for the Web

The results obtained from the comparative analysis served as the starting point for the design and implementation of our event detector. This analysis helped us in identifying the key aspects that should be considered when designing and implementing such systems for a distributed environment like the Web. In this section we present the most important aspects that we have considered during the implementation of our notification system.

7.1.1 Event Languages

We need to design appropriate event language for defining the different types of events relevant for an application. In some cases, event languages may already exist, e.g. as it is the case with the event algebra defined in XChange. In such cases we could either extend the language in order to obtain a more expressive one or use it without modifications.

When modelling reactive applications we may have *high-level, application-dependent events* (defined by application-dependent domain ontologies). We may also have *low-level events* that reflect modifications on the database or repository level; these ones are generally associated with data manipulation operations executed on the database. Moreover, we may have *events defined by application-independent* domain ontologies, which represent happenings in application-independent domains. In addition to this, events can be classified into *atomic* and *composite* events. Therefore, the ontology of events and the expressivity of the language, i.e. the type of events that can be defined with the expressions of the language, must be considered when selecting or designing an event language. In other words, depending on the type of events we want to detect, an event language could be more appropriate than others.

The specification of atomic and composite events requires different event lan-

guages. In the case of composite events, an event algebra providing operators for combining simpler events is needed. The event algebra may include different types of operators depending on the type of composite events we need to define. for example, temporal combinations of simpler events require operators that consider the time relations between events. As for atomic events, an event language should provide constructs that match the update operations of data repositories. For example, for detecting insertions of data in XML repositories, the event language used for specifying such events should provide a construct matching that operation, e.g. *insert(Fragment1,Fragment2)* to denote the insertion of XML *Fragment1* as a child of *Fragment2*.

7.1.2 Event Detection Semantics

When detecting composite events, we need to differentiate between the event's occurrence time and the event's detection time. An event's occurrence time is the point in time at which the event effectively occurs, whereas the event's detection time denotes the point in time at which an event detector detects the event. For atomic events, *occurrence* and *detection time* coincides; however, for composite events these are different.

Two possible semantics for detecting composite events can be considered. The *detection-based semantics* and the *interval-based semantics*. With the *detection-based semantics* composite events are detected at the end of the interval over which they occur. In other words, the detection time of a composite event corresponds to the detection time of the last constituent event that has been detected. Under this semantics the event's occurrence and detection times are considered the same. On the other hand, the *interval-based semantics* detects composite events over an interval. It considers the starting and ending point of the interval over which a composite event occurs, thus the event's occurrence time differs from the event's detection time.

The problem of detection-based semantics is that it does not capture the intuitive meaning of some event expressions. However, as pointed out in [4], interval-based semantics is not appropriate for all situations either and hence, both semantics should be considered depending on the situation at hand.

7.1.3 Techniques for Detecting Events

As we have seen, a considerable number of strategies for detecting events have been proposed in both the database and the Web field. The choice of the appropriate strategy is influenced by factors such as its inherent complexity, the storage requirements for data structures, the types of events to be detected (atomic or composite) and the context where events occur (databases, XML/RDF repositories, etc.). Therefore, strategies for detecting events can be divide into two groups: those used for detecting composite events and those for detecting atomic events.

A strategy for detecting composite events must implement the operational seman-

tics of an event algebra. More specifically, it must provide a set of data structures used for storing event data and a detection model. A detection model specifies how simpler events (atomic or composite) are combined in order to form composite events. It considers the semantics of the event algebra's operators as well as the restrictions (e.g. temporal restrictions) imposed to the set of constituent events. In general, two alternatives exist for implementing detection of composite events: *non-incremental* and *incremental evaluation* approaches. Due to its efficiency, the incremental evaluation of event expressions is preferred. However, a variety of techniques for implementing incremental evaluation of event expressions have been proposed in the literature. The three most common ones are based on: *Petri nets*, *Finite State Automata* and *Event Trees*.

- *Strategy based on Petri nets* Petri nets are a simple but powerful formalism for modelling sequential and concurrent system behaviour. They are a suitable mechanism when efficiency in the memory usage is important. The main advantage of using Petri nets is that, as with the tree-based approach, they allow the implementation of optimization techniques for event expressions. Common (sub)expressions within a single event expression or among different event expressions can be reused by combining single Petri nets. Moreover, rewriting techniques can be implemented in order to transform an event expression into an equivalent one. However, Petri nets are inefficient compared to other approaches; as pointed out in [47].

- *Strategy based on Finite State Automata* The approach based on Finite State Automata is suitable for detecting composite events whose expressions are equivalent to regular expressions. It works fine when primitive events have no event parameters and thus, no parameter computation is required. However, if primitive events include parameters, this technique requires further analysis in order to ensure an efficient implementation. Additionally, this strategy allows the implementation of optimization techniques for event expressions thus, reducing memory usage and improving the efficiency of the overall event detection process.

- *Strategy based on Event Trees* The tree-based approach is a relatively simple and flexible mechanism for implementing composite event detection. It provides a more natural way to represent an event expression as its structure is mirrored in an event tree; thus, facilitating the debugging process and being easier to understand by humans. One of its advantages is its flexibility to support the implementation of a variety of tasks; besides event detection. Some event detectors have implemented mechanisms for deleting registered event expressions, deleting event instances from the event trees or, as done in our prototype, checking whether or not an event expression may still produce event instances. Another important advantage is that by using trees, a large collection of efficient algo-

rithms for manipulating trees become immediately available. Algorithms for traversing, constructing and erasing trees can be easily reused and adapted. Furthermore, event trees support the implementation of optimization techniques for event expressions. For example, the combination of event trees into event graphs helps to exploit commonalities within the same expression or among different expressions; thus, improving the event detector's overall performance.

Although these techniques have been widely used for implementing composite event detection in the different domains, we believe that the tree-based approach provides the greatest benefits. The relative inefficiency of Petri nets and the difficult to easily model event parameters with FSA; together with the simplicity and flexibility provided by the tree-based approach, makes the later more appropriate than the others. However, it is important to notice that other techniques may exist that are more efficient or appropriate in some cases. Therefore, a detailed analysis of the available possibilities should be conducted at implementation time; taking into account the peculiarities of each case.

As for atomic events, the strategy used for detecting them largely depends on the type of atomic events being detected and the domain where they occur. Detection of atomic events in Active Databases is relatively easier compared to event detection in XML/RDF repositories and efficient techniques based on the monitoring of update operations exist. This is due to the relatively richer set of atomic event types found in the XML/RDF domain and the semi-structured nature of such documents, which makes events hierarchically related. In the context of XML and RDF documents, techniques based on the monitoring of repository's update operations, the use of DOM events and the comparison of different versions of the same document have been proposed. The DOM Event Model supports the detection of mutation events (modification to XML documents), although this is not enough in some situations [19]. Strategies based on the monitoring of repository operations are suitable for situations where documents are only modified through the repository. They represent a simple but efficient approach to event detection. However, when documents can be modified outside the scope of the repository, which is very common in the Web, these strategies are not appropriate and thus, an approach based on versions comparison and *edit-script events* is required.

7.1.4 Representation of Events

An event is an abstract concept. Therefore, in order for events to be processed, detected and exchanged by systems they must be represented using some sort of format. In other words, the information associated with an event, i.e. its event data, must be represented in such a way that systems are able to process it. In distributed environments such as the Web, an event that occurs at one location (e.g. detected by one event

engine) might be processed at another locations; by e.g. an ECA rule engine or another event detector. Thus, the information associated with such an event needs to be represented so it can be used by the receiver. Event data includes the values of the event parameters as well as any additional information regarding the occurrence of the event, like e.g. detection time, occurrence time and type of event.

Two aspects of event representation are important: an *event's internal representation* and an *event's external representation*. An event's internal representation is determined by the data structures used by a system (ECA engine, event detector, etc.) so as to represent the event. Here, several alternatives exist depending on the programming language used for implementing the system. Most systems are implemented following the object-oriented paradigm and so they use objects to represent events. In this case, event parameters are usually represented by an object's attributes. Notice, however, that additional data structures might be required in order to assist the event processing in a system. On the other hand, an event's external representation determines the way event data is formatted in order to allow the exchange of events. In this case, and considering an heterogeneous and distributed environment such as the Web, an XML-based data format is the more suitable approach to event representation. XML not only supports the exchange of events in a meaningful way but also the extraction of data from the event representation; XML Query languages could be easily used for extracting such information. This is the approach followed by active languages such as [7,63]. In these approaches both events and variables are marked up using an XML-based Markup Language.

7.1.5 Extracting Information from Events

Events communicated between systems serve the purpose of carrying information (event data) related to a situation that has occurred. For example, an event reflecting the insertion of a tuple in a table may include as part of its data the name of the target table. In order for an event detector to process event data, there must be a method for extracting this information from an event's representation. That is, an event detector must be able to access the event parameter whose value represent the target table.

In Active Databases this is usually implemented by using a pair of system variables. The database's state before and after the execution of a data manipulation operation is usually stored variables `OLD` and `NEW` respectively. The same mechanism have been implemented in some active languages for the Web, such as [10,22,58]. This approach works fine in centralized environments, where the event detector is integrated into the active system. However, in distributed and heterogeneous environments such as the Web a different approach is required. The Active languages [7,63] use a mechanism similar to the one used in Logic Programming. Event data is communicated among systems and ECA components by using *logical variables* (variable bindings). These variables are instantiated or bound by e.g. an event detector and then used by other com-

ponents in order to access the information regarding the event that has just occurred.

Logical variables are used for implementing both horizontal and vertical communication in the context of ECA rules. That is, communication between rule components and communication between the ECA engine and other engines, like e.g. event detectors and query processors. Moreover, logical variables can be bound to several things, e.g. literals, XML/RDF fragments, events, or URIs.

7.1.6 Communication of Events

Regarding exchange of events, we agree with the idea that events should be exchanged between system according to a *message-based* communication model and a *push strategy*. In contrast to a *pull strategy*, where reactive systems receive and detect events by periodically posting queries on other systems (Web sites), we consider that a *push strategy*, whereby events are detected at one location and then sent (pushed) to interested locations by sending messages, contributes to reducing network load and supports faster event detection; i.e. events are detected as soon as they occur.

7.2 Future Work

Regarding the implementation of our event detector, there exist several aspects that we would like to consider in the future:

- *Type system for variables.* The current prototype does not take into account the type of variables. Using types we can implement additional checks when detecting events. Here, as proposed in [7], an ontology specifying a type system may be used as it allows for sharing of values.
- *Computation of variable bindings.* The approach used in XChange for computing variable bindings seems to be adequate for this event detector. In principle, it would be possible to invoke a constraint solver in order to compute the variable bindings. However, a deep analysis of the performance of this and other alternatives should be made.
- *Expressive power.* Although the language implemented by the current prototype is suitable for a wide range of applications and is expressive enough to analyze different aspects of event detection, we would like to extend its expressivity. In this sense, the implementation of cumulative operators (SNOOP language) would provide a higher expressivity, extending in this way the applicability of the evaluator. Also, other parameter contexts could be implemented.
- *Composite event detection semantics.* We would also like to examine the implementation of *interval-based semantics* for composite event detection.

- *Detection window.* The current prototype does not consider the time event instances stay in the system. Depending on the rate at which instances (atomic events) arrive to the event detector this may be a problem. The use of a detection window ensures that event instances are eventually discarded from the event engine, reducing in that way the storage requirements. In this sense, we would like to explore different alternatives and then implement a detection window.
- *Optimization of event expressions.* An event can be specified by several equivalent expressions. In this sense, rewriting techniques may simplify the detection process and uncover common (sub)expressions. Therefore, optimization of expressions is an important aspect of the implementation as the event detector is expected to receive and process an important number of event expressions.
- *Dealing with delays.* In distributed environments such as the Web, delays in the network communications may cause out-of-order event delivery and thus, incorrect event detection. Therefore, a strategy for dealing with late notifications should be considered.

Bibliography

[1] Apache Xindice. http://xml.apache.org/xindice/index.html.

[2] eXist - Open Source Native XML Database. http://exist.sourceforge.net/.

[3] Xcerpt: A Rule-Based Query and Transformation Language for the Web. http://www.Xcerpt.org/.

[4] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.

[5] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[6] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, 1995.

[7] José Júlio Alferes and Wolfgang May. Specification of a model, language and architecture for reactivity and evolution. Technical Report IST506779/Lisbon/I5-D4/D/PU/a1, August 2005.

[8] José Júlio Alferes and Gastón Tagni. Implementation of a Complex Event Engine for the Web. *IEEE Services Computing Workshops (SCW'06)*, 0:65–72, 2006.

[9] Ricardo Amador. Resourceful Reactive Rules. http://rewerse.net/I5/r3/index.html.

[10] Angela Bonifati and Daniele Braga and Alessandro Campi and Stefano Ceri. Active XQuery. *ICDE*, 00:0403, 2002.

[11] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Patranjan. Flavours of XChange, a Rule-Based Reactive Language for the (Semantic) Web. In *RuleML*, pages 187–192, 2005.

[12] James Bailey, George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for xml. In *Web Dynamics*, pages 223–248. 2004.

[13] James Bailey and Alexandra Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *DBPL*, pages 252–270, 1999.

[14] James Bailey, Alexandra Poulovassilis, and Peter Newson. A dynamic approach to termination analysis for active database rules. In *Computational Logic*, pages 1106–1120, 2000.

[15] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. Analysis and optimisation of event-condition-action rules on xml. *Computer Networks*, 39(3):239–259, 2002.

[16] Elena Baralis, Stefano Ceri, and Stefano Paraboschi. Improving rule analysis by means of triggering and activation graphs. In *Rules in Database Systems*, pages 165–181, 1995.

[17] Elena Baralis and Jennifer Widom. An algebraic approach to rule analysis in expert database systems. In *VLDB*, pages 475–486, 1994.

[18] Elena Baralis and Jennifer Widom. An algebraic approach to static analysis of active database rules. *ACM Trans. Database Syst.*, 25(3):269–332, 2000.

[19] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite events for xml. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 175–183, New York, NY, USA, 2004. ACM Press.

[20] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language, W3C Proposed Recommendation. `http://www.w3.org/TR/xquery`, November 2006.

[21] Angela Bonifati. *Reactive Services for XML Repositories*. PhD thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, December 2001.

[22] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active Rules for XML: A New Paradigm for E-Services. *VLDB*, 10:39–47, 2001.

[23] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. In *WWW10*, Hong Kong, China, March 2001.

[24] François Bry, Michael Eckert, and Paula-Lavinia Patranjan. Reactivity on the web: paradigms and applications of the language XChange. *Journal of Web Engineering*, 5(1):003–024, March 2006.

[25] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *ICLP*, pages 255–270, 2002.

[26] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165, 1999.

[27] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[28] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Enginering*, 14(1):1–26, 1994.

[29] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997.

[30] SS. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. ACM SIGMOD Conference*, pages 493–504, Montreal, June 1996.

[31] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., 1999.

[32] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and querying changes in semistructured data. In *ICDE*, pages 4–13, 1998.

[33] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. pages 26–37, May 1997.

[34] James Clark. XSL Transformations (XSLT): Version 1.0, W3C Recommendation. `http://www.w3.org/TR/xslt`, November 1999.

[35] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[36] World Wide Web Consortium. Document Object Model (DOM) Level 2 Events Specification, November 2000.

[37] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*.

[38] Michael Eckert. Reactivity on the Web: Event queries and composite event detection in XChange. Master's thesis, Institute for Informatics, University of Munich, Germany, 2005.

[39] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`, Octubre 2004.

[40] Stella Gatziu and Klaus R. Dittrich. Samos: An active object-oriented database system. *IEEE Data Eng. Bull.*, 15(1-4):23–26, 1992.

[41] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using petri nets. In *RIDE-ADS*, pages 2–9, 1994.

[42] N. Gehani, H. Jagadish, and O. Shmueli. *Advanced Database Concepts and Research Issues*, chapter Compose: A system for composite event specification and detection. Lecture Notes in Computer Science. Springer Verlag, 1994.

[43] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

[44] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[45] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an Active Object–Oriented Database. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 81–90, San Diego, California, 1992.

[46] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[47] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 11–20, New York, NY, USA, 1990. ACM Press.

[48] Anton P. Karadimce and Susan Darling Urban. Refined triggering graphs: A logic-based approach to termination analysis in an active object-oriented database. In *ICDE*, pages 384–391, 1996.

[49] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *11th Intl. World Wide Web Conference (WWW2002)*, 2002.

[50] Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-Based Java-Scripting for a Bioinformatics Semantic Web. In *DILS*, pages 17–30, 2004.

[51] M. Liu, L. Lu, and G. Wang. A declarative XML-RL Update Language. In Springer-Verlag, editor, *Int. Conf. on Conceptual Modeling (ER 2003)*, pages 506–519, 2003.

[52] Masoud Mansouri-Samani and Morris Sloman. GEM: A Generalized Event Monitoring Language for Distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[53] Wolfgang May, J. J. Alferes, and Ricardo Amador. A General Language for Evolution and Reactivity in the Semantic Web. In François Fages and Sylvain Soliman, editors, *Principles and Practice of Semantic Web Reasoning PPSWR'04*, volume 3703 of *LNCS*, pages 101–115. Springer, 2005.

[54] Wolfgang May, J. J. Alferes, and Ricardo Amador. An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3761 of *LNCS*, pages 1553–1570. Springer, 2005.

[55] D. Mishra. SNOOP: An event specification language for active databases. Master's thesis, Database Systems R and D Center, CIS Department, University of Florida, Gainesville, FL 32611, August 1991.

[56] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. In *IEEE Proceedings - Software*, volume 148, pages 1–10, 2001.

[57] G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-Condition-Action Rules on RDF Metadata in P2P Environments. In *Proc. 2nd Workshop on Metadata Management in Grid and P2P Systems (MMGPS): Models, Services and Architectures*, Senate House, University of London, December 2004.

[58] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proceedings of the 3rd International Workshop on Web Dynamics (in conjunction with WWW2004)*, 2004.

[59] G. Papamarkos, A. Poulovassilis, and P. T Wood. WP4 deliverable 4.4. ECA rule languages for active self e-learning networks. Technical Report IST-2001-39045, School of Computer Science and Information Systems, Birkbeck, University of London, January 2004.

[60] G. Papamarkos, A. Poulovassilis, and P. T Wood. Event-Condition-Action Rules on RDF Metadata in P2P Environments. *To be published in Elsevier Computer Networks*, October 2006.

[61] Norman W. Paton, editor. *Active Rules in Database Systems*. Springer, New York, 1999.

[62] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[63] Paula-Lavinia Patranjan. *The language XChange: A declarative approach to reactivity on the Web*. PhD thesis, Institute for Informatics, University of Munich, Germany, 2005.

[64] Sebastian Schaffert. *A rule-based query and transformation language for the Web*. PhD thesis, Institute for Informatics, University of Munich, Germany, 2004.

[65] Daniel Schubert. Development of a prototypical event-condition-action engine for the Semantic Web. Bachelor Thesis, University of Gottingen, 2005.

[66] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, 2004.

[67] Serge Abiteboul and Dallan Quass and Jason McHugh and Jennifer Widom and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[68] S.Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *First International Workshop on Rules in Database Systems*, Edinburgh, August 1993.

[69] S.Gatziu, A. Gepert, and K. R. Dittrich. Integrating active concepts into an object-oriented database system. In *Third International Workshop on Database Programming Languages*, Nafplion, August 1991.

[70] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD 2001: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 413–424, 2001.

[71] A. Buchmann U. Chakravarthy M. Hsu R. Ladin D. McCarthy A. Rosenthal U. Dayal, B. Blaustein and S. Sarin. The HiPAC project: Combining active databases and timing constraints. *ACM-SIGMOD*, 17(1):51–70, March 1988.

[72] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[73] Wolfgang May. XPath-logic and XPathLog: A logic-programming style XML data manipulation language. *Theory Pract. Log. Program.*, 4(3):239–287, 2004.

[74] World Wide Consortium (W3C). XML Path Language (XPath). `http://www.w3.org/TR/xpath`, November 1999.

[75] World Wide Consortium (W3C). RDQL - A Query Language for RDF (W3C Member Submission), January 2004. `http://www.w3.org/Submission/RDQL/`.

[76] World Wide Consortium (W3C). SPARQL Query Language for RDF (W3C Working Draft), October 2006. http://www.w3.org/TR/rdf-sparql-query/.

[77] XML:DB Initiative. XUpdate - XML Update Language, September 2000. http://xmldb-org.sourceforge.net/xupdate/.

[78] Robert J. Zhang and Elizabeth A. Unger. Event Specification and Detection. Technical Report TR CS-96-8, Department of Computing and Information Sciences, Kansas State University, June 1996.

Appendix A

Class Diagram

Figure A.1 on the next page depicts the set of classes used for implementing the prototype. In the diagram, class `Evaluator` is defined in the R3 Java library. Classes `EventTree`, `EventTreeNode` and `OperatorNode` implement the functionality of an event tree. In this case, classes `OperatorNodeOR`, `OperatorNodeANY`, `OperatorNodeS` and `OperatorNodeAND` implement the semantics of the event language's operators.

The class `EventDetectionEngine` implements the core functionality of the detector by invoking the different modules of the system.

The class `AtomicEventProvider` implements a "dummy" event broker that signal event instances to the system.

Figure A.1: Class Diagram