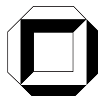
Universität Karlsruhe (TH)
Research University · founded 1825

Complex Event Processing in AJAX-Applications with Rule Engines

Diploma Thesis
Prof. Dr. Rudi Studer
Institute of Applied Informatics
and Formal Description Methods AIFB
Universität Karlsruhe (TH)

of

cand. inform.
Roland Stühmer

Supervisors:

Prof. Dr. Rudi Studer
Dr. Ljiljana Stojanovic (FZI),
Kay-Uwe Schmidt (SAP Research CEC Karlsruhe)

Started on: January 01, 2008
Submitted on: July 14, 2008

Address:
Roland Stühmer
Hans-Sachs-Str. 1
76133 Karlsruhe

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, ______________________________

Zusammenfassung, deutsch

Aktuelle Rich Internet Applications (RIAs) benutzen Techniken wie die *Comet*-Architektur oder das Abrufen von RSS-Tickern als grundlegende Infrastruktur um Ereignisse auf den Web-Client zu propagieren. Jedoch bieten solche RIAs keine Möglichkeiten, um die Ereignisse weiterzuverarbeiten. Diese Lücke soll mit dem in dieser Diplomarbeit vorgeschlagenen Ansatz geschlossen werden. Der Ansatz kombiniert die Verarbeitung komplexer Ereignisse und die Ausführung von Regeln mit einem ontologiebasierten Modell direkt im Web-Client. Ein Rahmenwerk wird vorgestellt, gebaut auf deklarativen Geschäftsregeln, um das Programmieren von reaktiven und anpassungsfähigen RIAs zu erleichtern. Die Arbeit besteht aus einer neuen *event-condition-action* Regelsprache, die auf die Anforderungen von RIAs zugeschnitten ist, sowie einem client-seitigen Regelprozessor, der in der Lage ist, komplexe zusammengesetzte Ereignisse zu erkennen und Regeln auszuführen.

Abstract

State of the art event-driven Rich Internet Applications (RIAs) rely on technologies like the Comet architecture or polling strategies like RSS feeds to provide a basic infrastructure for transporting events to the Web client. However, such RIAs provide no means for further processing of these events. This gap will be closed by the approach proposed in this thesis. The approach combines complex event processing and rule execution with an ontology-based object model directly on the Web client. A framework is proposed, built on declarative rules for achieving reactive and adaptive RIAs. The work consists of a novel event-condition-action rule language tailored to the needs of RIAs as well as a client-side rule engine capable of detecting complex events and executing rules.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xv**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Goals . . . 3
1.3 Methodology . . . 3
1.4 Results . . . 4
1.5 Thesis Structure . . . 5

2 Related Work **7**

2.1 Event Processing . . . 7
2.1.1 Complex Event Processing and Event Stream Processing . . . 8
2.1.2 Event Specification Languages . . . 9
2.1.3 Event Detection Algorithms . . . 13
2.2 ECA Rule Languages . . . 15
2.3 Logic Programming Approach to Event Detection . . . 16
2.4 Rich Internet Applications . . . 17
2.5 The Rete Algorithm . . . 18
2.6 Ontologies and Knowledge Representation . . . 19

3 Analysis **21**

3.1 Requirements for Event Detection . . . 21
3.2 Requirements for Condition Matching . . . 25
3.3 Requirements for Rule Actions . . . 26
3.4 Requirements for the Rule Language . . . 28

4 Design **31**

4.1 Complex Event Processing . . . 31
4.1.1 Snoop as the Event Detection Algorithm . . . 32
4.1.2 Simple Events . . . 34

4.1.3 Event Operators 35
4.1.4 Event Graph 37
4.2 Condition Matching 43
4.2.1 The Rete Network 45
4.2.2 Adding Objects 47
4.2.3 Removing Objects 48
4.3 Ontology 49
4.4 Client-side Rule Language 51
4.4.1 Event Part 54
4.4.2 Condition Part 57
4.4.3 Action Part 62
4.5 Overall Architecture 63

5 Implementation 67
5.1 JavaScript Language Features 67
5.2 JavaScript Object Notation 70
5.3 Comet Architecture 71
5.4 Graph-based Complex Event Detection 72
5.5 Rule Engine using Rete 75

6 Summary and Conclusions 77

A Grammar of the JSON Rule Language 79

Bibliography 87

Index 93

List of Figures

4.1 Event Graph Example 38
4.2 Event Detection (Class Diagram) 39
4.3 Triggering of the A(E1, E2, E3) event operator (Activity Diagram) . 40
4.4 Working Memory (Class Diagram) 44
4.5 Rete Network Example 45
4.6 Rete Network (Class Diagram) 46
4.7 Tradeable Ontology (Asserted Hierarchy) 50
4.8 EventRIA Main Class (Class Diagram) 63
4.9 System Overview (Component Block Diagram) 64

List of Tables

2.1 Conclusions from Event Specification Languages and their Implementations 14

3.1 Requirements for Event Detection 24

3.2 Requirements for Condition Matching 26

3.3 Requirements for Rule Actions 28

3.4 Requirements for the Rule Language 30

List of Abbreviations

AJAX	Asynchronous JavaScript And XML
ANTLR	ANother Tool for Language Recognition
BNF	Backus-Naur Form
BRA	Business Rules Approach
CEP	Complex Event Processing
DOM	Document Object Model
ECA	event-condition-action
EDA	event-driven architecture
ESP	Event Stream Processing
JSON	JavaScript Object Notation
LHS	left-hand side
OWL	Web Ontology Language
RHS	right-hand side
RIA	Rich Internet Application
WME	working memory element

1

Introduction

This diploma thesis establishes a framework for Rich Internet Applications. The framework conveys advantages of enterprise application development to the client side of Web programming. This introductory chapter explains motivation, goals and methodology involved in reaching these goals. In the end an outline of the remaining chapters is given.

1.1 Motivation

The setting for this work is Rich Internet Applications (RIAs). RIAs are Web applications, executing a large part of their application logic locally in the client, e.g. in a Web browser. Communication with the Web server is asynchronous, non-blocking, using Asynchronous JavaScript And XML (AJAX). Therefore, such applications are more responsive to the user's input than classical request/response, page-based Web applications.

Extended possibilities of the user interface, like drag and drop, also set RIAs apart from traditional Web applications. Possessing these characteristics of desktop applications is what motivated the name *Rich Internet Application*.

RIAs came into existence with the rapid increase of end-user bandwidth in the Internet, as well as growing client-side computing power and evolving Web browser capabilities, for example new features of the client-side JavaScript programming language.

Unfortunately, RIAs are growing fast in complexity. They are dealing with business applications which were formerly the sole domain of server-side execution. Examples are client-side generation of graphs and charts from raw data, handling of events which are streamed to the client and thus integrate the client in a distributed event-driven architecture (EDA), and a plethora of other examples which rely on client-side computing power on the one hand and on the other hand require creativity as well as hard work by the developer.

Apart from these developments in RIAs, powerful rule-based systems have been part of enterprise applications on the server side for a while. Rule-based systems offer a declarative approach to programming. In conjunction with a high-level view on business objects, this ideally allows non-programmers to create applications, apart from the advantages outlined in the following paragraphs.

Rule-based systems also provide a high grade of flexibility in changing business environments. Because rules are loosely coupled, e.g. in comparison to imperative code, they are easily altered, interchanged or amended. This provides an adaptability which is otherwise not possible in more static approaches to programming.

Another advantage of the rule-based approach are the possibilities of code reuse. First, rules are per se succinct, standalone pieces of application logic. Thus, they offer a predetermined granularity to simplify reuse. Second, they are specified in their rule language as self-contained entities. Therefore, they are readily packaged for exchange. Third, because rules have a declarative, implementation-independent view on programming, their reuse in different systems is simplified.

Complex Event Processing (CEP) is a second feature of enterprise application development, which would aid the development of RIAs. It is closely related to rule-based programming since the advent of reactive rules in the 1980s [DaBM88]. Today RIAs handle events from the user interface or streamed/polled from the internet. By subscribing to events across a network such an application might take part in a larger event-driven architecture.

The set of events triggered by a certain subsystem might be well defined by the author author of the system. However, the events may not fit the application-specific needs of a particular RIA, or of requirements evolving over time [BrEc06]. Therefore, it must be possible, even in client-side applications, to infer more events and information from a given set of events.

CEP strives to find complex events from “smaller” constituent events. Because complex events only exist implicitly, they must be mined from streams, and matched to patterns. Knowledge about events is thereby refined or increased.

1.2 Goals

The overall goal of this thesis is to make RIAs more reactive and adaptive. Following the ideas from Section 1.1 this goal is manifold. Four main sub-goals can be identified as follows.

The first goal is to enable client-side applications to adaptively handle events, simple and complex. This implies the ability to find complex events among streams of simple events. Furthermore, adaptivity of the event detection will create the possibility to modify the instructions for this process during run-time. Reaching this goal will make client-side applications more flexible than just being able to react to raw events with pre-set granularity and semantics.

The second goal is to enhance adaptivity for the rest of the Web application. Apart from adaptivity of the event detection, the state-changing logic will also be adaptable. This regards the parts of the application concerned with context, rather than events. In doing so, the complete logic of an application will be adaptable.

The third goal is to foster declarative programming for Web applications. Reaching this goal will help capitalising on the experience of non-IT professionals who are accustomed to authoring business rules for server applications, and possibly to reuse pre-existing business logic from server rules.

The fourth goal is to integrate access to the work done for the preceding goals. Thus, a uniform point of access should be presented, in the form of a reaction rule language. Doing so, will enable a Web developer to harness the whole approach presented in this thesis.

1.3 Methodology

The goals are found by observation of the demanding development of growing client-side applications. Literature on server-side developments is studied to find requirements and related solutions for event detection and condition matching. Finally RIA-specific requirements are collected.

The goals are reached by choosing a suitable existing event specification language and extending it to mitigate some observed drawbacks. Also, existing rule languages are compared to find inspiration for the new language, fulfilling all collected requirements, especially for Web programming needs.

In the implementation phase several workarounds for common shortcomings of the JavaScript programming language have to be found. This process further provides motivation to facilitate more declarative programming in Web applications.

1.4 Results

The contribution of this thesis is threepart. The results want to promote reactive, adaptive Rich Internet Applications. They are listed as follows.

- As a first contribution it will be shown how to accomplish detection of complex events in RIAs. State-of-the-art event frameworks for RIAs like [CCBF07] still require a centralised server to process events. The approach shown in this thesis will achieve complex event detection entirely on the client side. This will entail the implementation of a data-driven, graph-based detection algorithm to infer complex events from streams of simple or less complex events. To infer complex events, a set of rules (i.e. event operators) will be presented, extending the set of Snoop [CKAK94] event operators. Extensions to these operators are made to provide further user-extensibility and access to a business vocabulary in the form of an ontology. An implementation of the detection algorithm including all event operators will be given.

- The second contribution of this thesis will show a client-side rule engine for RIAs. The rule engine will be comprised of a Rete [Forg82] implementation for the Web with some optimizations specific to Web programming. Also, the client-side ontology may be used in the rule predicates just as in the event operators. As to the knowledge of the author this will be the first implementation of Rete or a similar algorithm for RIAs, providing a rule-based approach to programming for Web developers.

- The third contribution will be the specification of a lightweight, extensible reaction rule language. The language will combine event expressions and conditions under the event-condition-action (ECA) paradigm. The language will be tailored specifically to the needs of RIAs and Web developers, being easily parseable on a Web client and providing language features appropriate and helpful for Web programming. The formal grammar of the language will be included in this thesis. Currently, no rule language exists for programming RIAs. This particular goal will attempt to fill that gap.

- As a fourth contribution it will be shown that parts of an ontology can effectively be used in RIAs. Means to access the ontology are provided for the event detection as well as the rule engine.

Together these results constitute a framework to create and run event-driven reactive and adaptive Rich Internet Applications. The framework shall encourage a declarative rule-based programming style. An implementation of the framework is presented.

1.5 Thesis Structure

The remainder of this work is outlined as follows: Chapter 2 contains terms and techniques which are the basis for this thesis. A large part of it is research having been conducted in enterprise application development, but for the server-side. Examples are CEP, the Business Rules Approach (BRA) and ontologies.

Chapter 3 defines requirements and sets goals for CEP, BRA, and the rule language to accommodate both. The goals are to guide the design process.

Chapter 4 is concerned with the design of the proposed solutions. The chapter describes the creation of the architecture for the different parts comprising this work; mainly the event detection, the forward-chaining rule engine and the grammar for the rule language. The chapter attempts to be independent of programming language specifics.

Chapter 5 presents details of the implementation, how the system architecture is rendered into the JavaScript programming language, and programming language-specific considerations. Problems and solutions on the implementation level are outlined.

Chapter 6 presents the summary of this work, drawing conclusions about what has been achieved and provides an outlook on possible future research and extensions of this work.

2

Related Work

This chapter discusses related work that has been conducted on fields of reactive systems, as well as rule evaluation. These fields mainly contain advances in enterprise application development which are not available for Rich Internet Applications yet, such as event processing for adjustable reactivity and rule-based programming for various reasons. The chapter finishes with a short introduction to ontologies.

2.1 Event Processing

Event processing is concerned with receiving events and refining them according to the specifications of subscribers. Events are created at *event sources* and are consumed at *event sinks*, i.e. the subscribers, [LuSc07]. Between the sources and the sinks events are processed to filter event information or gain new information, which is returned in the form of further events.

Events are used to trigger actions at their subscriber's sides. Events thus govern control flow of applications, providing *virtual synchrony* in distributed systems. This means that systems run in perceived synchrony although control is relayed in a message-passing kind of way, asynchronously. This may involve delayed network links, possible buffering of messages, etc. Nevertheless, mere virtual synchrony greatly eases the design of distributed systems because many aspects of the system may be treated independently without affecting the overall correctness [BiJo87].

Event sources provide *simple events*. A simple event is a notable incident detected inside or at the boundaries of a system. The event is called simple because it is not

composed or abstracted from other events. The event is made explicit by creating a first class object [Luck01]. The object holds the type of event and the time of detection as a minimum. Optional further values are the origin or system component of detection and further data depending on the type of event. The manifested event subsequently functions as a message which can be passed along to different parts of the system.

Historically, most of the work in event processing has been originating from database management systems with the goal of making databases *active*. An active database is a database which can act on events from inside or outside the system. Events in active databases occur on data manipulation, as well as on timers or from external sources. For relational databases this includes *insert*, *update* and *delete* operations. For transactional databases, the start and end of transactions are also of interest. Object oriented databases add method calls. Simple events are triggered for these actions. Timers in the system contribute temporal events.

As mentioned before, events consist of a type, their occurrence time and possibly other parameters. All the parameters can be used in data analysis to help in detecting larger event patterns. Building on the event types one can create complex nested expressions, using a set of operators. Examples are: `And`, `Or`, `Sequence` and others. The result is an algebra of events and event operators. Sets of events which match a given expression comprise a complex event. In turn the pattern expression yields the type of the complex event.

Apart from operators like `And`, `Or`, `Sequence`, etc, there are many others which have been proposed since the start of Complex Event Processing (CEP) in the early 1990s. An overview of existing operators is given in Subsection 2.1.2, after the following section briefly compares CEP and ESP.

2.1.1 Complex Event Processing and Event Stream Processing

Complex Event Processing and Event Stream Processing (ESP) are two fields of research concerned with handling events. Traditionally they tried to solve different problems in event processing using different approaches. The next paragraphs will outline these approaches, as they are described e.g. by David Luckham in [Luck06] and by others.

Events may happen in a stream or in several streams (a cloud). The field of ESP is concerned with extraction of events from a stream. Thus, ESP handles events that are totally ordered by time. Further emphasis of ESP lies on efficiency for high throughput and low latency. Processing is done by analysing the data of the events and selecting appropriate occurrences. Long-running queries produce results regularly; an analogy of ESP may be drawn to signal processing.

CEP, on the other hand, is more focused on complex patterns of events. To detect these patterns CEP takes more time and memory than ESP. CEP is concerned

with clouds of events, which yield only a partial temporal order of events. Other partial orders of interest for CEP are causality, association, taxonomy, ontology. Rather than to signal processing, an analogy of CEP may be drawn to higher-level situational inferencing.

However, CEP and ESP nowadays adopt each other's approaches. The two worlds become mingled and authors such as Bass [Bass07] propose to consider them as one.

2.1.2 Event Specification Languages

This subsection describes several noteworthy event specification languages along with their drawbacks and advantages. The following Subsection 2.1.3 then describes associated implementations which are often closely connected with their specification languages. Table 2.1 on page 14 gives a concluding summary.

Early event specification languages were developed for active databases. They use complex event specifications to facilitate database triggers which do not only listen to simple events but observe complex combinations of events to react upon.

Complex event specifications are patterns of events which are matched against the streams of events that occur during the system run-time. These patterns consist of simple event types and event operators. Simple events are the basic events the system can detect. Complex events are derived from occurrences of one or more of them.

All simple events have a simple event type, which for a database application might be *insert*, *update* and *delete*. The types are used as placeholders in event patterns.

Event patterns are structured by event operators. A given operator might have several event types as arguments and e.g. stipulate that the constituent events must occur in sequence. An event detector for the given pattern functions as a stream pattern matcher and listens for events that satisfy the type constraints and together satisfy the semantics of the given operator, e.g. occurred in sequence.

Many operators were proposed in the past and the following paragraphs discuss several event pattern languages and their operators. Usual operators offered by many languages include disjunction, sequence and accumulation.

One early active database system is HiPAC [McDa89]. It is an object-oriented database with transaction support. As stated in [ZiUn99], HiPAC provides the basic three event operators of disjunction (`E1 OR E2`), sequence (`E1;E2`) and accumulation (`*E1`). The disjunction pattern matches if an event of one or the other type occurs. The sequence pattern matches if events of the specified types occur in sequence. The cumulative pattern matches once at the end of a transaction interval if at least one event of the specified type has occurred. Thereby, several occurrences are accumulated in the single complex event.

HiPAC can detect events only within a single transaction. Global event detectors are proposed which detect complex events across transaction boundaries and over longer intervals, but no further details are given.

Ode [GeJS92] is another active database system with a language for the specification of event expressions. The language is also referred to as Compose. Ode proposes several basic event operators and a large amount of derived operators for ease of use and shorter syntax. Basic operators are as follows. The conjunction operator (`E1 AND E2`) matches if one of each event types occurs. The operator `!E1` matches any single event that is not E1. This is not to be confused with the Not operator from Snoop which matches the non-occurrence of E1 within a specified interval, regardless of other occurrences during that time. Snoop is described later in this section. The remaining basic operators from Ode are: `relative` and `relative+`. The operator `relative(E1, E2)` describes the occurrence of the two events in sequence and `relative+(E1)` describes one or more occurrences of E1.

In addition to these basic operators are many derived operators. A few examples follow. The operator `prior(E1, E2)` describes two events that occur after one another, with no overlap allowed. The operator `sequence(E1, E2)` describes two events that immediately follow one another, with no other events occurring in-between. The different sequencing operators relative, prior and sequence may be specified with an arbitrary number of arguments, e.g. `sequence(E1, ..., En)`. On top of the structural event operators, Ode possesses a so-called event mask, which is a predicate an event must fulfil in order not to be discarded. Masks may be used to filter events of a certain type to adhere to further constraints.

The last of the classical event specification languages discussed here is Snoop [CKAK94] and its successor SnoopIB. Snoop offers the previously described operators `And`, `Or`, as well as `Sequence`. The remaining operators are `Not`, `Any`, `A`, `A*`, `P`, `P*` and `Plus`. They are briefly explained as follows. Event `Not(E2, E1, E3)` is detected at the end of the interval [E1, E3], if no event of type E2 has occurred during the interval. Event `Any(m, E1, ..., En)` is detected when m of the specified participants have occurred. Also, Snoop defines a so-called aperiodic operator `A(E1, E2, E3)`. Event A is signalled each time E2 occurs within the time interval started by E1 and ended by E3. There also is an associated cumulative event A*, which is signalled at the end of the interval and is comprised of all occurrences of E2 from the interval. Event `P(E1, TI[:parameters], E3)` occurs for every time interval TI, starting at E1 and ceasing at E3. The list of parameters is collected each time P occurs. If not specified, only the occurrence time of P is collected by default. In an object-oriented database a parameter specification is a function which is then invoked after every TI and its result is added to the event parameters. The respective cumulative version of P is expressed as `P*(E1, TI:parameters, E3)`. Unlike P, it occurs only once at the end of the interval. Specified parameters are collected and accumulated. They are made available when P* occurs. Note that the parameter specification is mandatory for P* as the default parameters from P, i.e. the accumulated sequence of periodic occurrence times, would in itself not be useful.

The last Snoop operator is `Plus(E1, n)`. It is used to specify a relative time event. E1 might be any (possibly complex) event and n is an amount of time. That time is added to the occurrence time of E1 to receive the occurrence time of the Plus event.

Early work on Snoop views events as having instantaneous occurrences [CKAK94, Section 2.2]. This is the case for complex events with constituents spanning an interval of time. As a result only the time of detection is used for the occurrence, instead of the interval from the start of the first constituent event to the end of the last constituent event. Consideration of only the time of detection is termed *detection-based semantics*. It poses problems with nested sequences as pointed out in [GaAu02].

Contrary to an intuitive understanding the expressions `E1;(E2;E3)` and `E2;(E1;E3)` are equal with detection-based semantics. The order of the events E1 and E2 in the example is inconsequential, because the whole event is detected if the event in front of the parentheses is detected before E3. No relation between events in position one and two is specified. Both expressions are fulfilled by the same event histories: `e1,e2,e3` and `e2,e1,e3`. This is not expected from a sequential relation. *Interval-based semantics* solves these problems by viewing complex events as occurring over the interval from the occurrence of the first constituent event, the initiator, to the end of the last constituent event, the terminator. Interval-based semantics for Snoop is called SnoopIB and was first published in 2006 as [AdCh06].

Selection and consumption of events define which occurrences participate in a complex event. Both terms are an integral part of the semantics of an event definition. *Selection* defines the choice of events if there are more than one event of a required type that have not yet been consumed. *Consumption* is concerned with the deletion of events when they cannot be part of further complex events.

Selection and consumption are, however, application specific and cannot be universally defined. An example illustrates this: For an event specification `E1 AND E2` a large number of events of type E1 may have been detected at a certain point in time. Once an occurrence of type E2 is subsequently detected, it must be decided which occurrence or occurrences of type E1 to pair with E2 to form the conjunction event.

Publications of Snoop identify four classes of applications which have prototypical needs concerning event selection and consumption. These are called *contexts*. Specifically they are called Recent, Chronicle, Continuous and Cumulative context.

The first type of application requires the Recent context. Such an application might be concerned with sensor readings or similar data where each occurrence of an event (such as E1 from the previous example) refines or updates a certain value. The most recent occurrence will be the one of interest. Older occurrences of E1 are deleted. The latest occurrence is retained and is not consumed by complex events which use it.

The second context is Chronicle context. It pairs the oldest matching events and immediately consumes them in the process. Applications using Chronicle context

require causal dependencies between events to be maintained, such as e.g. the start and end of transactions, bug reports and their related product releases.

The third type of application requires Continuous context. In this context, a given terminator causes the detection of all currently initiated events. The terminator and the initiators are consumed. This behaviour facilitates applications that monitor trends, for example the changes of a stock price at the end of a fixed period in different time windows started by the different initiators.

The fourth context is termed Cumulative context. It collects all occurrences of participating event types until the terminator is detected. All participants are consumed. This context enables applications to collect all actions of specified types in a system up to a deadline, represented by the terminator.

Although the four detection contexts do not claim to be complete, they are often used by further research, for example in recent publications of Reaction RuleML [PaKB07]. The contexts defined by Snoop laid the foundation of reducing the numbers of possible combinations of events from the unrestricted context, in a well-defined way. *Unrestricted context* is the complete history of all events. In this context, an event pattern `E1 AND E2`, for example, results in the Cartesian product of the histories of E1 and E2. The reduction in combinations of events is done by identifying the aforementioned typical application needs, and clearly defining them by means of initiator and terminator events.

Arbitrary policies for event selection and consumption are possible in some related works. Two approaches can be found in [ZiUn99, HiVo02]. They allow fine-grained specifications for detection and consumption of participant events for a given event expression. However, they do not supply any reusable patterns, like the predefined contexts defined by Snoop. In the future these approaches might provide a fundamental event processing algebra according to [Etzi08], like the relational algebra for relational database theory. Unfortunately this is out of the scope of this thesis.

Other approaches to event pattern languages include statements reminiscent of SQL. Two examples are StreamSQL[1] and Continuous Computation Language CCL[2]. Queries in these languages match patterns in streams instead of database tables. Queries are long-running and produce incremental results in contrast to SQL queries. In CCL sliding windows are supported. Joins are possible to form complex events and patterns may be specified using the operators conjunction, disjunction, sequence and negation. All operators can only be applied to bounded windows of events. Complex events have to adhere to SQL schemata which prohibits nested sets, for example an events that includes a previously unknown number of constituents. Although the well known syntax of SQL might help with the adoption of these languages, a seamless integration of an action part seems hard to accomplish.

[1] StreamSQL Guide. Online Resource. http://www.streambase.com/developers/docs/latest/streamsql/

[2] Coral8 CCL Reference, Version 5.2. Online Resource. http://www.coral8.com/system/files/assets/pdf/5.2.0/Coral8CclReference.pdf

None of the above mentioned approaches interact with the business vocabulary. Simple event types have only names. Complex event types in HiPAC, Ode and Snoop only have structure but no names. Embedding these types in an overarching ontology enhances the scope of event processing by providing precise understanding of information carried by events and by allowing to integrate concepts from existing business objects. An ontology of events could relate event occurrences to the actions in the system which caused them (e.g. business process ontologies). Moreover events are related to each other, creating hierarchies of composition or modelling causal relationships, etc, which can be represented by ontologies. Also, ontologies may allow formulating business rules that are more user-comprehensible by using a shared vocabulary which is well known to the users, i.e. rule authors.

Table 2.1 on the next page contains an overview of identified drawbacks of specification languages. Please note that the table additionally contains combined conclusions from the following subsection on detection algorithms as well. So those results are added to what has been discussed this far.

2.1.3 Event Detection Algorithms

Many of the aforementioned event languages belong to their respective database management system, or prototype thereof. Three of them have noteworthy implementation details: The Ode approach conducts complex event detection by using automata. SAMOS uses coloured Petri nets. Sentinel uses a graph-based approach.

Complex event detection in Ode [GeJS92] is implemented using automata. Input for the automata is a stream of simple events. Thus, Ode transforms complex event expressions into deterministic finite automata. For sub-expressions which are complex events themselves, the process is done recursively. Atomic simple events are ultimately represented as automata with three states; a start state, an accepting state, entered upon detection of the simple event occurrence, and a non-accepting state, entered upon detection of any other simple event. Apart from providing the design for an event detector, automata are a convenient model to define semantics of complex event operators. As a downside an automaton cannot accept overlapping occurrences of the same complex event. Also, event parameters pose a problem. They are either stored outside of the automaton, or the automaton is increased greatly in the number of states to accommodate the different parameters and possible values thereof.

Complex event detection in SAMOS [GaDi94] is implemented using Petri nets. Each primitive event type is represented by a Petri net place. Primitive event occurrences are entered as individual tokens into the network. Complex event expressions are transformed into places and transitions. Where constituent events are part of several expressions, duplicating transitions are used to connect the simple event with the networks requiring it. This results in a combined Petri net for the set of all event expressions. Petri nets, like automata provide a model of the semantics of

event operators. Also, the detection of overlapping occurrences is possible. Event parameters are stored in tokens and flow through the network. Although the tokens are individual there is no mechanism to deterministically choose a token if more than one are in a single place.

Table 2.1: Conclusions from Event Specification Languages and their Implementations, summarised from Subsection 2.1.2 to Subsection 2.1.3 on pages 9–13.

Topic	Open Issues	References
HiPAC	• detection restricted to a transaction • only detection-based semantics • no content-based checks	[McDa89]
Ode/Compose	• nondeclarative mix of event and condition • automata do not detect simultaneous complex events of one type • automata cannot easily handle event parameters • only detection-based semantics	[GeJS92]
SAMOS	• no clear strategy for event consumption • only detection-based semantics • no content-based checks	[GaDi94]
Snoop/SnoopIB	• no content-based checks	[CKAK94], [Chak97], [AdCh06]
StreamSQL, CCL	• not extensible for actions as in ECA • detection only within fixed windows	Vendor Web sites
ADL	• nondeclarative mix of event and condition	[Behr94]
Reaction RuleML	• reactive semantics simulated by polling	[PaKB07]

Sentinel [Chak97] is an active object-oriented database implementing complex event detection for the Snoop operators. Event detection follows a graph-based approach. The graph is constructed from the event expressions. Complex expressions are represented by nodes with links to the nodes of their sub-expressions, down to nodes of simple events. Event occurrences enter the bottom nodes and flow upwards through the graph, being joined into composite occurrences. The graph is a directed acyclic graph and generally does not form a tree for two reasons: nodes may have several parents, when their represented expression is part of more than one complex events, and furthermore, there is no single root node when there is no overarching, single most complex event.

A possible drawback of Snoop compared to the previously mentioned implementations is that the data structures of Snoop do not represent and even clarify the semantics of the event expressions. The logic of Snoop is hidden in the implementation of each graph node. However, the semantics of Snoop is defined externally, using event histories and describing the operators as mappings from simple event histories to complex event histories. Furthermore, Snoop defines the selection and consumption of simple events for the concurrent detection of overlapping complex events. The four alternative definitions, Recent, Chronicle, Continuous and Cumulative context were described earlier.

This concludes the discussion of event detection algorithms. Table 2.1 on the facing page contains a summarised overview of the different specification languages and their algorithms, outlining identified drawbacks of the different approaches. This will later give reason to choose Snoop over the other languages and implementations but also to make an extension to Snoop concerning the current lack of content-based event operators.

2.2 ECA Rule Languages

Many active database systems not only specify an event language but also a reaction rule language. Reaction rules or event-condition-action (ECA) rules complement the event specification with a condition to be evaluated upon event occurrence and a corresponding action to take. The term ECA rule was first used in conjunction with the HiPAC active database [DaBM88]. ECA rules are a generalization of several methods to achieve active behaviour, such as triggers and production rules, which had been in prior existence but treated separately.

The ECA rule approach divides the rule execution in three parts, event, condition and action handling. The parts are processed concurrently but work with different input [BrEc06]. Event processing deals with transient input, i.e. the events. Although events are objects, they are usually not made persistent but are used in an on-line, real-time fashion to deduce complex observations and thereby consume the less-refined input. The output from the event detection is knowledge about complex distributed incidents happening in a system. This knowledge incorporates the information from individual events in an accumulated fashion. Conditions, on the other hand, deal with persistent data or knowledge. They may be viewed as queries to a database or a knowledge base. Although the outcome of conditions may change over time, the condition evaluation generally deals with long-lived data, which unlike events may not be time-dependent.

Many reaction rule languages have been proposed in the past. Not all obey the separation of event, condition and action specifications. Computational issues have been identified early. Large amounts of memory and computational effort may be wasted on the detection of events which are subsequently discarded when corresponding conditions are not met. This led many language designers to dilute the event and

condition parts. Contrary to a declarative approach it is then up to the rule author to revise events expressions for run-time efficiency.

Ode [GeJS92] for example, promotes so-called EA rules, instead of ECA rules, where the conditions are folded into the event specification. This is done by mixing event expressions with filters. These filter predicates can impose conditions on events in order to discard occurrences early. Thereby, the deleted occurrences do not use further resources or become part of complex events which will not be needed.

Similarly the logical language ADL (Activity Description Language) [Behr94] has (EC)*A-rules. Like the event masks in Ode the language tries to achieve finer granularity by mixing event and condition specifications.

Reaction RuleML [PaKB07] is a language for extended ECA rules, with an XML representation. The specifications of event, condition and action are strictly separated. More precisely rules are expressed as tuples `(T, E, C, A, P, EL)`, consisting of T time, E event, C condition, A action, P post condition and EL contingency action, "else". Parts might be left blank, i.e. always satisfied, stated with "_", e.g. , `(T, E, _, A, _, _)`. Blank parts might be completely omitted, leading to specific types of rules, e.g. standard ECA rules: `(E, C, A)`.

Event specifications of Reaction RuleML follow the Snoop [CKAK94] operators, redefined with an interval-based semantics. Different consumption "policies" are mentioned but do not seem to adhere to the Snoop consumption modes Recent, Chronicle, Continuous and Cumulative.

Each rule part of Reaction RuleML is formulated in logic programming. This results in a homogeneous rule language which at the same time satisfies the demand for declarative expressiveness. The implementation of the logic interpreter, however, might cause problems with the event detection, as pointed out in the following Section 2.3.

2.3 Logic Programming Approach to Event Detection

To satisfy the demand for declarativity in their rules, many rule languages choose logic programming as a paradigm. Prolog interpreters are readily available to provide well-understood implementations. Moreover Prolog supplies precise semantics to any rule language which is reduced to Prolog logic.

A straightforward idea for a rule-based system is then to use the same system for events, just as for Prolog facts. Incoming events are added to Prolog's fact base when they occur. This provides a homogeneous handling of events along with other facts, e.g. existing business objects.

To verify a logic predicate, Prolog uses a backward-chaining algorithm. This means a query-driven, top-down approach. At the time of issuing a query, the resolution

tries to find valid sub-predicates of the query (sub-goals) which contribute to the verification or falsification of the complete predicate.

Backward-chaining therefore requires someone to “ask a question” first, (issuing a query), before any action is taken. This does not resemble a reactive event-driven system. New events can only be detected at query-time, there is no spontaneous, direct reaction to an incoming event.

To mimic reactive behaviour, a rule may query for its observed events at regular intervals. The previously mentioned Reaction RuleML does exactly this: To simulate active behaviour in its backward-chaining inference engine, Reaction RuleML employs a polling layer on top of the inference engine to continuously emit queries for new events. Apart from using computing resources on polling, this merely emulates true reactive semantics [ScAS08].

The approaches previously mentioned in Subsection 2.1.3 on page 13 are data-driven, thus forward-chaining. This means that a bottom-up approach is used to reach goals. New events are fed into an algorithm trying to verify bottom level goals, then triggering higher goals until the overall goal of a rule is reached. This approach has two main advantages. Firstly this way of reasoning yields truly reactive systems, secondly it is better equipped to process large amounts of data (i.e. incoming events) because no search with backtracking is required over the entire fact base.

2.4 Rich Internet Applications

As mentioned in Section 1.1, Rich Internet Applications (RIAs) are applications running on a Web client. They exhibit a rich user interface comparable to native desktop applications, can perform intensive calculations on the client but at the same time communicate in a non-blocking asynchronous fashion with servers across the Internet. This combination shall facilitate responsive, desktop-like applications on the client.

The term Rich Internet Application itself was first used in [Alla02], a Macromedia white paper on their proprietary FlashMX product. The mentioned product only runs in Web browsers using a plug-in, however, the above mentioned definition of a RIA also applies to applications written in JavaScript. They do not require any plug-ins or other installation effort on Web clients. This thesis is focused on the latter type of RIAs.

Work from [CCBF07] proposes an event-driven architecture for RIAs. Events are monitored on the client and may originate from user interaction or from remote event sources. However, for complex event processing the events are sent to a server.

The approach proposed in this thesis handles complex event processing directly in the RIA and thereby reduces latency and communication cost. Additionally the work from this thesis goes further than event processing and embeds the event part in a

complete rule language executable on the client. The rule language not only provides reactivity to events from different sources, local and remote, but also monitors state in objects from a working memory using the Rete algorithm.

2.5 The Rete Algorithm

Rule-based systems offer a declarative approach to programming by formulating the business logic in a set of rules. Such rules are high-level language expressions concerned with the state of business objects. The state is declaratively described in the left hand side (i.e. condition) of a production rule, containing a condition and a consequential action.

Production rules are traditionally used in conjunction with a working memory. A working memory is the potentially large set of values or (business) objects which may be referenced in the conditions of rules. Rule actions are fired when a condition becomes true.

However, conditions can be expensive to evaluate; a large working memory requires a lot of elements to be taken into consideration, and complex conditions require nested checks to be performed. Once a single element in the working memory is added, changed or deleted, the conditions of many rules may change their outcome. In the worst case every rule must be re-evaluated.

The Rete algorithm is a forward-chaining algorithm for evaluating production rules. It is designed by Charles Forgy and described in [Forg82]. The Rete algorithm remedies the situation of re-evaluating rule conditions by introducing state-saving of the evaluation process between changes to the working memory. The condition evaluation needs not to be fully recomputed.

This is accomplished by dividing the conditions into a hierarchical network of nodes, each doing a single comparison, filter operation, join, etc. This represents a partial match of the condition. Every node has a so-called memory which stores the objects that fulfil the constraints of its node. When an object is changed, the network does not need to be completely refilled, but only the changed object is re-fed into or removed from the network. A classic Rete network is divided into two parts; the first, called the alpha network, contains nodes with one input edge. These nodes perform filter operations using single conditions or constraints. The second part, the beta network, contains nodes with two input edges, joining objects from two subordinate nodes. A third type of nodes exists which models the top nodes. The rule actions are invoked from these nodes.

The Rete algorithm has two successors: Rete II and III, but they are not published. There are other optimised algorithms based on Rete; TREAT [Mira90] and LEAPS

[Bato94] being two examples. Variations on Rete are implemented in many up-to-date rule engines, for example ILOG JRules[3], jBoss Drools[4], Jess[5], etc.

2.6 Ontologies and Knowledge Representation

> *An ontology is an explicit specification of a conceptualization.*
> —Thomas R. Gruber [Grub93]

In other words, an ontology[6] formally describes features and relationships of concepts. It may be used to represent knowledge for sharing and reuse. Conceptualization of knowledge is achieved using classes and relations.

Classes represent real-world entities from a certain domain which is to be described in an ontology. Relations are parameters which a class and its instances have. The most powerful application of relations is referencing other classes. Thereby, relationships are established forming a network of classes and their relations.

Ontologies are encoded using ontology languages. These are formal, logic-based languages. They obey their respective grammars and are the groundwork for exchanging and reusing the created conceptualizations.

Ontologies enable applications to commit to a shared view on things and thus improve interoperability. Practically this means agreeing on using a certain vocabulary in a way consistent with a theory specified by an ontology [Grub93].

For the client-side rule engine presented in this work, a service is used which prepares ontologies for use on the Web client. The service processes ontologies specified in Web Ontology Language (OWL) and performs some ontology reasoning on the server. A set of data structures is returned from the service which are used by the client-side application to answer certain queries to the ontology.

---

[3] ILOG JRules. Online Resource. http://www.ilog.com/products/jrules/

[4] Drools. Online Resource. http://jboss.org/drools/

[5] Jess. Online Resource. http://herzberg.ca.sandia.gov/

[6] "The term is borrowed from philosophy, where an ontology is a systematic account of Existence." [Grub93]

3

Analysis

This chapter defines the requirements for event detection, condition matching and the rule language. The requirements are later used in Chapter 4 for the design of the event-enabled rule engine.

3.1 Requirements for Event Detection

The user of an event detector is a rule author, who needs to define a complex event pattern, e.g. for a reactive rule. Such a user wants to abstract and simplify information, hidden in the event stream.

The implementation of the event detector must put into effect the specifications from the user. Suitable algorithms and data structures are needed which can accommodate all user requirements. Additionally, non-functional requirements are added to that. The interests of the user are discussed first.

Complex event processing is a problem of pattern matching. The user defines the patterns. In order to find events with meaningful event patterns, the user is primarily concerned with the power of expressiveness an event pattern language provides. An event pattern language is an abstract algebra (cf. [Macl99]) over the set of all events, using operations of grouping, filtering, etc, to form expressions.

Events by themselves are notable incidents detected inside or at the boundaries of a system, cf. Section 2.1 on page 7. These incidents are materialized as first class objects. Events entering the system are referred to as simple events. As a minimum,

events carry their time of occurrence and their type, which is a class specifying the kind of incident being represented. Event types are application dependent, examples are *click* or *keypress* of a Web application, *create*, *update*, *delete* of a database application, *stockPriceChanged* or *doorOpening* of an algorithmic trading application or a building security system, respectively.

Complex events are detected from occurrences of simple events. The detection of a complex event usually takes a certain amount of time, for example when the complex event consists of two events occurring at two separate points in time. The occurrence of the complex event can then be recorded as the time of detection of the last constituent event, i.e. detection-based semantics. Alternatively it can be recorded as the interval of time spanning all constituent events, i.e. interval-based semantics. Subsection 2.1.2 discusses the disadvantages of detection-based semantics exemplified by the distributivity of the sequence operator. It is therefore a requirement for the event detection in this work to employ interval-based semantics for complex events.

Complex events are patterns of events which the user defines. Patterns can themselves be input for further patterns. Patterns are formed by event relations or event operators, as they are referred to in this work. The operators an event detector exhibits are the building blocks for the user to work with, for building expressive, nested patterns of events.

Event operators introduced above are divided into different categories. In Subsection 2.1.2 on event specification languages, a number of operators from different specification languages have been enumerated. They differ in their observation of events, focusing on various characteristics of events. Three distinct categories are outlined in the following paragraphs.

One category of operators is concerned with the mere presence or absence of events. A second category is concerned with temporal relationships of events. A third category is concerned with the contents of events, i.e. their event parameters. Each category itself forms a useful algebra over events which would enable meaningful event-driven applications. However, to form the most versatile applications, all aspects of events must be allowed to be included in event expressions.

The first category consists of logical operators. They employ boolean operations on the occurrences or non-occurrences of events. Among them should be `AND`, `OR`, `NOT`, which the user must be able to nest, as appropriate. Boolean operators are part of every event specification language since the beginning of of research on complex event detection. Examples treated in Chapter 2 are HiPAC, Ode, SAMOS, Snoop and others.

The second category consists of temporal operators. For these operators the mere presence of an event does not suffice. Temporal constraints are imposed on participating events to fulfil a pattern. Such constraints might e.g. include the sequence of two or more events, occurrences of one event type within an interval formed by two others, etc. Operators of this kind are part of Ode, SAMOS, Snoop and others.

The third category consists of guards, i.e. filters. These operators must be able to perform content-based matching, looking inside the events. The structural operators from the previous categories do not include this functionality per se. Guards are unary operators, thus they do not provide structure. They rather operate like filters, thinning out a single stream of events, according to one or more conditions. Conditions should be able to flexibly check all event parameters. The term content guard is used by David Luckham for this type of event operator [Luck01, p. 167], the term event mask by the authors of Ode [GeJS92].

Combining the three categories results in an event algebra which covers logical, temporal as well as data-driven approaches to processing events. In order to form the most versatile expressions, all these aspects of events must be allowed to be included in an event detector.

Apart from the user's requirements for the exhibited operators, there are a number of additional requirements imposed by a practical implementation. An enumeration of such requirements is given as follows, starting bottom-up from the simple events.

Simple events must be flexibly implemented to fit all types of sources. Events may originate in many different systems for different reasons. They all should be handled because processing events from diverse sources greatly enhances the possibilities and scope of an event-driven application. Possible event sources for Rich Internet Applications are the following:

- User events occurring from user interaction with the Web browser and the Document Object Model (DOM).
- Temporal events created by the system clock such as time-outs, intervals, recurrence and offsets.
- Server-triggered events, received over the Internet. These events should be subscribed to via *pull* or *push* technologies [Mart99].

A general purpose event framework (in this case for Rich Internet Applications) must accommodate all kinds of events in a flexible or at least extensible fashion. Care should be taken as to a sufficient level of abstraction for representing simple events, so that the event detection can handle diverse types of events.

So far requirements for simple events are discussed. Respectively, requirements for complex events and their detection are collected in the following.

There are several algorithms which were proposed in prior research for event stream pattern matching. These are discussed in Subsection 2.1.3 about event detection algorithms; they use automata, Petri nets or a more general, graph-based approach.

The approaches have different advantages and disadvantages discussed in the section mentioned above. Requirements for the semantics of complex event specifications influences the decision for a suitable detection algorithm.

Complex events of a given complex type may occur simultaneously. This means several complex incidents of the same type happen at the same time in an overlapping fashion. Such overlapping events must not be ignored by the event detector.

As a motivation, e.g. a fraud detection system must be able to report different simultaneous occurrences of suspicious behaviour; Several occurrences of fraud share an event type, but that does not mean that a user is not interested in every individual report.

Complex events often have to be found among several suitable candidates for constituent events. Policies for deterministically choosing participating events are called event selection. For example, a conjunctive event of type `AND(E1, E2)` might have already detected several events E2 at a certain point in time. By the time the first event of type E1 is detected, it must be decided, which occurrences for E2 should be paired with E1 to form the `AND` event.

The topic of event selection is discussed in Subsection 2.1.2 on page 9 as well as the solution from Snoop, introducing four representative detection contexts Recent, Chronicle, Continuous and Cumulative.

Different modes of event selection are a way of defining the semantics of complex event specifications. It should at least be an optional requirement for an event detector.

Event consumption on the other hand specifies how constituent events are discarded. The latter may take place after events have become part of a complex event or when they cannot be part of any further complex events. Event consumption is closely related to event selection. Snoop treats both according to its detection contexts. Therefore, if an event detector provides different event selection policies, it should also provide policies on consumption.

Table 3.1: Requirements for Event Detection, summarised from Section 3.1

Requirement	must/should
Events are first class objects	MUST
Events contain time and type as a minimum	MUST
Interval-based semantics	MUST
Logical event operators	MUST
Temporal event operators	MUST
Content-based event operators	MUST
Sources: DOM, clock, remote	MUST
Detect overlapping complex events	MUST
Support selectable detection contexts	SHOULD

Apart from its primary effects on the semantics, consumption also has an impact on the performance of event detection. Different modes of consumption provide a

means of adjusting the run-time characteristics. This might be desirable for a rule author, i.e. user, having to obey strict processing- or memory constraints. Especially the amount of required memory is affected by the choice of consumption policy.

In summary, a good complex event detector relies on two things: Its rich set of operators and an efficient algorithm to evaluate these operators. Details concerning operators, events, and algorithms were given in this section.

3.2 Requirements for Condition Matching

Conditions are formulae over the state of an application. When a given formula is fulfilled, the system is in a state where the rule author wants some action to be executed. The system state is comprised of the memory contents. This is often simplified as a set of objects held in a so-called working memory. These working memory elements represent the system state as far as a rule system is concerned.

Traditional rule systems only execute CA rules. The systems are called production systems. Two examples are OPS5 [BFKM85] and CLIPS [CuRD93].

In the proposed rule-based framework an algorithm is needed to select and then execute the rules. Event detection for event-condition-action rules has been discussed above. The event detection is responsible for evaluating the event part, meaning that events are detected and notified. In order to assess which of the conditions are fulfilled, the condition part also requires a kind of "detection".

Event-condition-action rules are a superset of all rules containing event or condition parts or both, cf. Section 2.2 on page 15. ECA rules therefore integrate different types of rules, like triggers (EA rules), production rules (CA rules) and reaction rules (ECA rules). In the following, ECA refers to the superset where the event part or the condition part may optionally be left blank, and thereby also including all the types of rules mentioned above.

To evaluate most types of ECA rules, a separate condition matcher is required in addition to the event detection. This can best be observed from the fact that CA rules lack the triggering event specification, therefore another way must be provided to find and run any applicable rule.

Furthermore, any applicable rule should be found at the time when its condition is fully satisfied. This means that changes to the state of the application should immediately be reflected in the activation of rules. No query-driven semantics should be used for rule activation because it would restrict the capacity to act to only certain intervals at which queries are issued.

Instead of a query-driven (top-down) approach, a data-driven approach must be employed. A data-driven approach fulfils conditional predicates in a bottom-up way, also called forward-chaining. The advantage of forward-chaining evaluation is that for each change of state affecting a condition, the partial match is saved until it

can be further completed to form a complete match in the end. The process is data-driven, meaning that with each change of state it advances synchronously. Therefore, complete matches are reported immediately when they come into existence.

For rules containing an event part, on the other hand, there are also benefits from matching conditions by forward-chaining. As it is said above, a forward-chaining algorithm evaluates conditions incrementally. The state of the evaluation is saved. It is thus possible to to find all the fulfilled (and partly fulfilled) conditions at a given time, without further cost. So when a reaction rule containing all parts of event-condition-action is triggered repeatedly by its event during a time when the condition is fulfilled, then no repetitive computation has to be performed on behalf of the condition.

The subset of EA rules is the only case which does not require any support for condition matching. This set includes rules which behave as triggers, where an event is unconditionally bound to a successive action. Actions are discussed in Section 3.3.

Concerning expressiveness of rule conditions it is stated above that conditions are formulae over elements in the working memory. This is a requirement for this thesis. Production systems like the mentioned OPS5 and CLIPS allow conditions only in triple form. The working memory of these system is a set of triples as well, in the form subject-predicate-object. Examples are "car_no_1 is red" or "car_no_2 follows car_no_1". A possible condition would be "X is red, car_no_2 follows X". The rule system then tries to fulfil the condition with a suitable binding for the free variables from triples in the working memory.

For an object-oriented programming language the working memory elements must be objects. To specify formulae over objects, the conditions must allow checks on object attributes like "car1.colour == red" as well as variable definitions like "X.colour == red, X.follows == X". Several definitions of X in a condition must then be resolved by the rule engine to find a valid match.

Table 3.2: Requirements for Condition Matching, summarised from Section 3.2

Requirement	must/should
Forward-chaining implementation	MUST
Specification of patterns of objects from working memory	MUST
Patterns use features from object orientation	MUST

3.3 Requirements for Rule Actions

There are several requirements for the action part of rules. Depending on the programming styles which should be enabled by using the proposed rule framework, these requirements must include the following considerations.

First of all the rule engine should allow for the highest possible flexibility. Programming for Rich Internet Applications, this means that arbitrary JavaScript actions must be allowed. This enables the rule author to access functionality for which there is currently no declarative approach. For example, this includes DOM operations, which make up the major part of user-interface related JavaScript applications. Flexibility notwithstanding, arbitrary JavaScript actions also allow for future use of JavaScript or DOM additions which might be unforeseeable at present.

Apart from the imperative approach using JavaScript, a declarative approach should be supported. Traditional production systems like OPS5 [BFKM85] and CLIPS [CuRD93] offer rule-based declarative programming. In these systems the actions can alter the system state by only specifying modifications to objects. Such modifications include adding and deleting objects, as well as modification of remanent objects. Such a style of programming should be supported in the proposed rule framework.

As a third type of rule action it might be useful to explicitly feed a new event back into the system. Other rules would be able to react to such an event, just like an event from any of the other event sources mentioned in Section 3.1. Such an explicit event would be created in a rule action (consequence) and can be part of the event specification of other rules. A rule having events as input and also as output can be seen as a rule of event derivation, instead of a trigger resulting in some state-changing action.

From a pragmatic angle the question might arise, why the event specification of the first mentioned rule is not just included ("inlined") in the second rule in place of the explicit event. There are two answers why this might be useful. The first answer is that the first rule can contain a condition part. Therefore, its action is not only dependant on the event specification, which is supposed to be copied, but additionally on system state or context, evaluated in the condition. The resulting explicit event is therefore a temporal incident, detected in a certain context of the system. It is a noteworthy addition to the semantics of a rule system.

The second answer to the question of usefulness of explicit events is readability and reuse; For a rule author it might be helpful to split up large event expressions into several rules. These rules then trigger explicit events to be picked up by succeeding rules. Thereby, a detected partial event is propagated for further aggregation. This process might help the rule author to partition large event expressions into meaningful parts, improving readability of the rule system. Also, the explicit events which signal partial detections can be reused in other rules, encouraging reuse of the parts.

The latter type of rule action, i.e. the triggering of explicit events, shows that rules can feed back. However, it should be clear that all types of rule actions can cause feed-back, resulting in an alive system, having rules activating further rules. The action type of modifying objects, which above is mentioned secondly, creates feed-back when its modification influences the condition of another rule, and thereby activating it. The action type of arbitrary code, mentioned firstly, can cause side-

effects of events being triggered or conditions changing their outcome. Therefore, all action types may feed back control-flow and data-flow into the rule system.

Table 3.3: Requirements for Rule Actions, summarised from Section 3.3

Requirement	must/should
Execution of arbitrary JavaScript	MUST
Declarative approach using ASSERT/RETRACT/MODIFY	SHOULD
Explicit event approach	SHOULD

3.4 Requirements for the Rule Language

For managing the proposed reactive rule engine, this thesis proposes an appropriate rule language. The language must consider requirements specific to Rich Internet Applications. Syntax and semantics of the language must be described.

The semantics of ECA rules must be clearly defined. For the events, conditions and actions by themselves, this can be done separately, for example by reduction to their respective underlying languages. On top of that must be clarified: the relationships of the event and condition parts as well as their effect on actions.

The so-called coupling modes from early research on ECA rules in the HiPAC project, [DaBM88], point out different relationships between events and conditions. However, in all cases a condition is evaluated after an event has occurred. No mode is defined requiring conditions to be fulfilled *during* the entire occurrence of an event.

More recent works, e.g. in [Bers02], suggest a revised semantics for ECA rules. It is stated there that the complete condition of a rule has to be satisfied during its whole occurrence, i.e. from the beginning of the occurrence of the first constituent event up to the end of the occurrence of its last constituent event.

This understanding of event-condition-action rules conforms to the notion of interval-based semantics established for complex events. Interval-based semantics views an event as having a duration, instead of viewing it as just an instant at detection time. The duration lasts from the start of the first constituent event to the end of the last constituent event, cf. Subsection 2.1.2 on page 9.

Therefore, an accompanying condition should span the entire interval of the event duration. Downsides of not using interval-based semantics are pointed out in [GaAu02] and the aforementioned Subsection 2.1.2 for events, and in [Bers02] for conditions. In short, for events this includes unexpected results from transitivity of multiple sequence operators, and for conditions it includes possible matches with events, violating temporal axioms like matching system context in the future.

So much for the considerations about semantics of the rule language. Requirements for the syntax are discussed next. The functional requirements for the language syntax are to accommodate all features previously discussed, in a specification language for the user. Non-functional requirements, broadly spoken, are user-friendliness as well as browser-friendliness.

As for the functional requirements, the language must expose all user-adjustable features of the event detection, the condition matching and the different kinds of actions. These features have been mentioned in the previous sections and will be revisited and be elaborated on in Chapter 4.

The non-functional requirement of user-friendliness has several aspects which should be covered in this thesis. First of all the language should be extensible. This includes permitting the future use of JavaScript features which are not known today. Also, this includes the possibility of adding further operators for the event and condition part.

In addition to extensibility, some measures of reusability should be provided. For example, complex event expressions which are repeated in several rules should be made reusable at design-time. The user should have the possibility of creating a set of named event expressions. These predefined expressions can be incorporated into further event expressions of different rules. A method of reuse at run-time is described above in Section 3.3 with so-called explicit events but the method of reuse on a syntactical level is independently possible.

Methods of reuse should also be provided for condition expressions and possibly actions. For the latter it might be possible to offer a library of predefined actions. User interface patterns [Tidw06] might help in finding a meaningful selection of such actions to be provided for the rule author. However, such efforts might be out of the scope of this thesis and left for further research.

User-friendliness should also cover the run-time of the rule framework. One important requirement arises when a rule author wants to add and remove rules while the rule engine is running. This requirement is divided into sub-requirements for the different parts of the implementation, where rules are evaluated. Both the event detection and the condition matching algorithms must be able to alter their data structures in a coherent manner when rules are added or deleted from detection.

Lastly, on account of browser-friendliness there are also some non-functional requirements for a rule language. As far as the possible acceptance of a new rule language goes, it can be very important that the language closely fits the environment in which it is to be used. To accomplish this, the language should be lightweight, easy to deploy in a RIA and AJAX environment and should honour JavaScript programming practices, where possible.

David Luckham writes in his book on event processing [Luck01, p. 146], that an event language must be *expressive enough*, must be *notationally simple*, and *semantically precise* and must have an *efficient pattern matcher*. He says this about event

languages, but the preceding analysis for this thesis has shown that Luckham's requirements hold true for the condition part, just as they do for the event part.

Table 3.4 summarises the collected requirements for the rule language.

Table 3.4: Requirements for the Rule Language, summarised from Section 3.4

Requirement	must/should
Description of syntax and semantics	MUST
Definition of relationships between event, condition and action	MUST
Exposing all user-adjustable features of the rule engine	MUST
Addition and removal of rules at run-time	MUST
Extensibility	SHOULD
Reusability of event and condition expressions	SHOULD
Honouring JavaScript programming practices	SHOULD

4

Design

The design process of the proposed rule-based framework is shown in this chapter. Requirements were collected in the previous chapter, and are used here to make according design decisions. Previous work especially on Complex Event Processing (CEP) and condition matching is discussed above in Chapter 2 and will be referenced in this chapter. This chapter starts with CEP, then covers condition matching for rule execution, then continues with the integration of ontologies to access a business vocabulary. The chapter closes with the rule language, which must embrace all the previous parts and provide a unified, homogeneous interface for rule authors.

4.1 Complex Event Processing

This section covers the design of the Complex Event Processing engine. As a part of the rule engine it is detecting complex events which provide user-adjustable reactivity for the rules and thereby for applications employing the rule engine.

In the following subsections the choice for graph-based event detection is explained. Then the object-oriented design of simple events is described. The next subsection describes how to construct complex events. Constructing these is the main function of CEP, creating new events from exiting ones, and thereby gaining new information from the stream of simple events. Patterns to create complex events are described then, and lastly the design and of the detection graph is explained.

4.1.1 Snoop as the Event Detection Algorithm

For the design of an efficient complex event detector several alternative algorithms were proposed in the past. They differ in their detection approach, using either automata, Petri-nets or a graph-based approach. They also differ in their effects on the semantics of events they detect, and differ in general versatility. The three approaches are explained in detail in Subsection 2.1.2 to Subsection 2.1.3 on pages 9–13.

Snoop is chosen from the available approaches as a basis for the event detection in this thesis. Along with that, Snoop's operators are adopted with some extensions and with according extensions of the detection algorithm. Reasons for choosing Snoop are given after a brief introduction to the Snoop algorithm.

The graph-based approach from Snoop operates as follows. The algorithm starts out with a tree. For a given event expression a top node is created. Its children represent the sub-expressions. This is done recursively down to the leaves at the bottom of the tree, which represent simple events which are atomic.

At run-time simple events are fed into the tree. This is done at the leaves. The specific leaf is determined according to the type of the simple event. A leaf node subsequently notifies its parent of the new detection. The parent decides whether it can incorporate the newly detected event in a pattern. This might depend on its other children. If the parent finds a match it in turn notifies its parent. In this way events are propagated upwards in the tree. Top nodes are marked with some action to be taken on detecting the complex event.

The detection graph starts out as a tree. As a measure of saving space, Snoop allows sharing of sub-expressions between events. This results in nodes having multiple parents. Sharing is performed as follows. A new expression is added to the graph. A sub-expression of it is already represented by a node in the graph. When the new expression is added the pre-existing node is used in place of the sub-expression. Thereby, the sub-expression is reused. In the process the pre-existing child node receives an additional parent to be notified on detected events. If one complex event contains a sub-expression more than once, there are more than one paths through the graph from the top node to the mentioned sub-expression. Cycles are formed, and the graph is therefore not a tree in general.

Thus, the method of Snoop is called graph-based. However, for visualizing the algorithm it might still be helpful to view event expressions as having a *tree* representation. Also, the *parent–child* relationship from trees is adopted for the Snoop graph, as well as referring to the simple event nodes as *leaves*.

A reason for choosing Snoop over the other detection methods is that the graph-based approach allows for detection of overlapping complex events. This is a requirement from Section 3.1 and it rules out automaton-based event detection for use in this thesis. The reason for that is explained as follows.

Complex events of a given complex type may occur simultaneously. This means several complex incidents of the same type happen at the same time, in overlapping fashion. Automaton-based algorithms are not capable of detecting more than one instance of the same complex event at the same time.

This is an inherent drawback of how automata are used for event detection. As elaborated in [GeJS92, GeJS93] automata are constructed from regular expressions specifying event patterns. Transitions model accepted events in a given state. An initial state is created with transitions for initiator events, the initial constituent events. The transitions lead to further states, and so on, up to one or more accepting states, where the complex event is detected. The complex event is then defined as the sequence of transitions which were taken from an initiator to a terminator event.

When an automaton must accept overlapping complex events, the following happens. A suitable initiator changes the state of the automaton away from the initial state by using one of the transitions. The automaton will then be in a state which accepts constituent events to continue completion of the first complex event. There might be no transitions accepting initiators for further complex events, until the automaton is reset after completely detecting the first. Although there might be other transitions labelled with the initiator event type, these events will be incorporated in the first complex event as intermediate constituents. Other complex events are only started at the initial state.

In summary this means that overlapping complex events are ignored, because once an automaton is in the process of detecting a complex event, it is usually not in its initial state anymore, to start detecting a second complex event at the same time. Algorithms based on Petri nets and on graphs do not share this deficiency. An important drawback of the Petri net-based approach, however, is pointed out next.

Snoop's graph-based approach is chosen over a Petri net-based approach, because Petri nets do not support user-defined selection of tokens when a transition is fired. This means it cannot be predetermined by the user, which constituent events, i.e. tokens, are used when creating a complex event. Therefore, SAMOS does not provide configurable event selection policies in its Petri net-based approach, cf. Subsection 2.1.3.

Coloured Petri nets are introduced in [Jens92]. They allow tokens to be individually distinguished. *Arc expressions* at the transitions might accomplish event selection based on individual attributes. However, SAMOS uses colours only to model event parameters and to propagate these parameters through the Petri net.

As a side note, timed Petri nets [Ramc74] are no viable alternative to solve the problem. They attach duration values to transitions to model processes of different execution times. No handling of temporal data in tokens is achieved, which would be required to select events.

This concludes the major reasons for choosing the graph-based approach over automata or Petri nets. Automata cannot detect concurrent complex events and Petri nets do not offer a clear strategy for event selection.

The choice of detection semantics is the next important decision which has to be made on behalf of the event detection. The detection semantics are concerned with whether complex events are represented by an interval or only by a point in time. The preceding analysis for this work showed that a detection-based (point in time) semantics delivers unexpected results for certain operators, cf. Section 3.1. An example given is the sequence operator.

Snoop revised its semantics towards an interval-based view of events, called SnoopIB [AdCh06]. The same holds for other event detection system like Reaction RuleML [PaKB07]. Therefore, when Snoop is discussed in chapters on design and implementation, this means the to up-to-date algorithms of SnoopIB.

4.1.2 Simple Events

Before the design of the event graph is described in detail, a description is given of the simple events. They are first class objects representing incidents detected in and around a system, cf. Section 2.1. Requirements for the design of simple event objects are collected in Section 3.1. According to that, simple events must be generic enough to represent incidents from all event sources which are supported. The sources for this work are Internet sources for server-triggered events, explicit events from rule feed-back, the system clock for temporal events and the user interface, i.e. Document Object Model (DOM) for events from user interaction. The sources are depicted on the left-hand side of Figure 4.9 on page 64.

The most general attributes of a simple event are its name and its occurrence time. More information can be contained the so-called event parameters. The anatomy of a simple event is modelled in Figure 4.2 on page 39, at the bottom of the diagram, and is outlined as follows.

Simple events have a `name`. It is an attribute of the class `Event.SimpleEvent`. Names of simple events are identifiers like stockPriceChanged or doorOpened, etc. The attribute is not called “type” for reasons of consistency. An attribute of that name is used elsewhere to contain entities from an ontology and therefore has a different meaning. The name of a simple event is determined by the event source or a source adapter. The name is not changed later and it determines which places an event may take in a pattern.

The occurrence time of an event is modelled as two date attributes `t_start` and `t_detect`. They are both inherited from the abstract class `Event`. Events occurrences are described as durations because of using interval-based semantics. Simple events occur instantly, however, so for them both dates are identical.

Event parameters are stored in the attribute `parameters`. The attribute is organized as a hash map of key-value pairs to store parameters. Parameters are optional data. They may provide additional information about the observed incident.

Next in the group of event classes is `Event.TemporalEvent`. Although all events are temporal data, this class of events is termed temporal event, for example in

literature on Snoop [CKAK94]. This is because such events are only time-related, originating from a system clock and having no other causalities. The temporal event objects contain a `name` attribute where the instances of this class store their time specification, a so-called time string. Time strings represent absolute and relative times in a textual manner. Temporal events, their specification and detection is described below, together with the design of their corresponding graph nodes in Subsection 4.1.4 on page 37. Apart from their way of detection, temporal events behave like simple events and can be used in patterns just like them.

The third class of events is `Event.ComplexEvent`. This class forms all events which are detected from other events, which can be simple or complex. Complex events are defined by the structure and identity of constituent event. Therefore, complex events do not contain a name. They contain an array `members` to hold their constituents of the abstract class `Event`.

The preceding paragraphs so far discussed the design of classes for simple and temporal events. These events are models for indivisible incidents. The next section describes how to construct composite events. Constructing these is the main function of CEP, creating new events from other events, and thereby gaining new information from a stream of simple events. Composite events, or the more general term complex events, are constructed from the event stream matching patterns. The patterns are formulated by the user in order to satisfy a given information need. The specification of these event patterns is facilitated by a language of operators. The operators employed in this work are explained in the following section.

4.1.3 Event Operators

Different options for event specification languages were discussed in Chapter 2. For this thesis, the event language from Snoop is used with an extension from Ode. For both languages, see Subsection 2.1.2.

From the pattern-based languages described in Chapter 2, Snoop is the newest and most widely accepted. Reaction RuleML for example relies on the Snoop patterns for modelling its event part [PaKB07]. The SQL-like event languages which were also discussed, do not provide a rich set of operators. Event operations have to be split up into several query statements which process the event stream like a pipe. The pattern-based language of Snoop provides a more declarative approach to event specification. Moreover, the nested expressions from Snoop result in a more homogeneous rule language than a series of consecutive queries. Such queries would enforce a very different style of notation for the event part of the rule language, in comparison to the first-order logic used for the rule condition part.

Snoop provides its set of operators to create structured patterns of events using logical and temporal operators. As observed in the requirements analysis in Section 3.1 this set must be extended by content-based checks, which will also be described, after the Snoop operators.

All operators from Snoop are included in the design of this work. The operators can roughly be divided into logical and temporal operators, cf. Section 3.1. A third type of operators, content guards, are described afterwards.

The operators from Snoop are listed in the following paragraphs. Another introductory description is given above in Subsection 2.1.2. Semantics of the operators is given with their corresponding graph nodes in Subsection 4.1.4 when designing the event graph. Generally speaking, Snoop forms an algebra of structural event expressions consisting of operators over the set of event types. Event types may be names of simple events or themselves complex expressions.

The logical operators from Snoop are `Or`, `And`, `Any`, as well as `Not`. Operators And and Or are binary operators in the sense that they involve two operands. Operator Any is a generalized form the preceding ones. It accepts an arbitrary list of parameters and a parameter m, which specifies what number of events from the list must be detected to match the Any pattern.

The temporal operators from Snoop are `Seq`, `A`, `A*`, `P`, `P*`, as well as `Plus`. Operator Seq is the sequence of two events in time. Operators A and A* are ternary operators, detecting occurrences of one event type when they happen within an interval formed by the two other vent types. A* is the variant which collects all events and occurs only once at the end of the interval with all the collected constituents. P and P* are ternary operators as well, they also accept two events starting and ending interval, but the third parameter is a time expression after which the events occur periodically during the given interval. A function may be specified to collect event parameters for each periodic occurrence. P* is the cumulative variant which occurs only once, containing all collected constituents. P stands for periodic because of its metronome characteristics. A stands for aperiodic because the detected constituents occur at irregular times. The Plus operator accepts an event type and a time expression. The Plus event occurs after the specified event type has occurred and the specified time has passed.

The operators mentioned so far are the complete set from Snoop. Content-based checks are added to them in order to fulfil the requirement for filtering by event parameters.

Content-based checks do not provide structure as the previously described operators do. Content based checks filter streams of events, resulting in streams which contain only events matching a constraint check. Such checks are concerned with the parameters of events. The appropriate event operators are called *guard* by David Luckham or *mask* by the authors of Ode. This work will use the term mask.

The event mask is designed as an operator with one event input and a boolean function to be applied to the input. The value returned from the function decides about whether the input is accepted or discarded. An incoming event is accepted if the function returns `true`. When specifying a mask expression, the function itself may be selected from a set of predefined mask types. Moreover, the event masks in

this work are extensible in the way that the function may optionally be an arbitrary user-defined implementation.

All operators grouping events from more than one input must generally select events out of input queues. This is because two types of events which should be matched e.g. in pairs, generally do not occur in equal numbers. So one input queue retains instances and requires a selection when events from it must be used. The solution for this problem is user-defined selection and consumption strategies. The four strategies from Snoop which are used in this work are Recent, Chronicle, Continuous and Cumulative contexts. They are described in Subsection 2.1.2. Each operator must implement the different semantics and the rule author must select the desired one. For this work the default is determined to be the Chronicle context.

4.1.4 Event Graph

The event graph is used to detect patterns of events. Patterns are formed by nested expressions. The event graph is a network of nodes which represent event expressions. There are special nodes types for every event type. Incoming edges of a node originate in child nodes which represent sub-expressions. Simple event nodes have no incoming edges. Outgoing edges connect a node to its parent which makes further use of detected events. Detected events are propagated upwards in the network, starting with simple events which are fed into the graph at the simple event nodes. The propagation ends at top nodes which have no further parents. In these nodes events are extracted from the graph and are handed on to some action, which in the process discards the event. See Figure 4.1.

Event nodes may have more than one parent. This occurs when an event expression is used in several places of a pattern. The reused expression is then manifested only once in the event graph but outgoing edges are linked to all nodes where the expression is reused. All parent nodes are informed equally of detected events.

When a parent node is notified of detected events, the parent's method `trigger(triggeringChild, occurrence)` is invoked by the child, cf. class ComplexEventNode in Figure 4.2. The parameter triggeringChild is used to pass a reference to the child which is triggering its parent. Thus, the parent can decide which child supplies the newly detected event, meaning which place in the event pattern the event may fill. The second parameter, occurrence, is used for the event object, belonging to a subclass of Event.

If the parent cannot match its complete pattern on being triggered, the new event must be queued depending on whether it can be part of a future match. Queuing is done in the parent. Because different parents might have different consumption policies, they cannot share an incoming queue. Also, the parent nodes would possibly delete events which were not yet consumed by others.

A solution employing some sort of markers would not save any space compared to the solution of separate queues. To mark events as deleted by certain parent nodes,

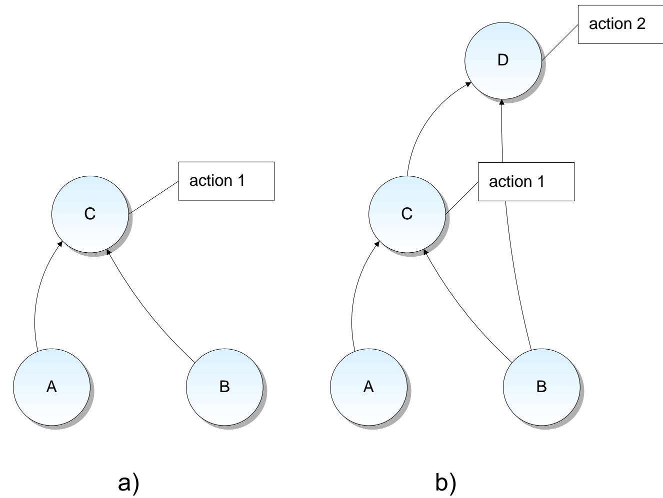

Figure 4.1: Event Graph Example. **Subfigure a)** shows an event graph with two simple event nodes A and B, and a complex event node incorporating both, e.g. `And(A, B)`. **Subfigure b)** shows the same graph with an additional enclosing complex event, e.g. `Seq(And(A, B), B)`. The node from the pre-existing sub-expression is reused. Subfigure b) shows that the event graph is generally not a tree.

each event in the (unified) queue would need a number of markers on the scale of the number N of parents. For M events in the queue this results in a space of $M \times N$ variables. However, since an event in a queue is only one reference, the duplication of queues for all parents also uses a space of $M \times N$. No space is saved by retaining a single queue. Moreover, the access time in the case of the unified queue is higher by a constant factor, for the time needed to check the marker.

Therefore, events are queued with their parents, which are then free to delete them upon consumption. Parents have different numbers of children, depending on the type of the parent. Operators `Or` and `And` e.g. have two children. The operator `Any` has an arbitrary number of children. Thus, a node has to queue incoming events for several children and thus the class `ComplexEventNode` is designed to contain the attribute `childrenQueues` which is an array of arrays of events, `childrenQueues: Event[][]`.

Graph nodes for complex event operators encapsulate the semantics of their respective operator. The semantics is implemented in the `trigger` method of each complex event node. Literature on Snoop defines the semantics of the operators using set operations on the ordered sets of events created by sub-expressions. These ordered sets are called event histories.

The semantics of event operators differ, depending on the detection context which is specified. The context may be seen as a filter for the event histories.

In an actual application of the Snoop algorithm, it is not desirable to store the entire history of events. Therefore, depending on the event context, a trigger method discards events which cannot be used in any further complex events. As it is mentioned

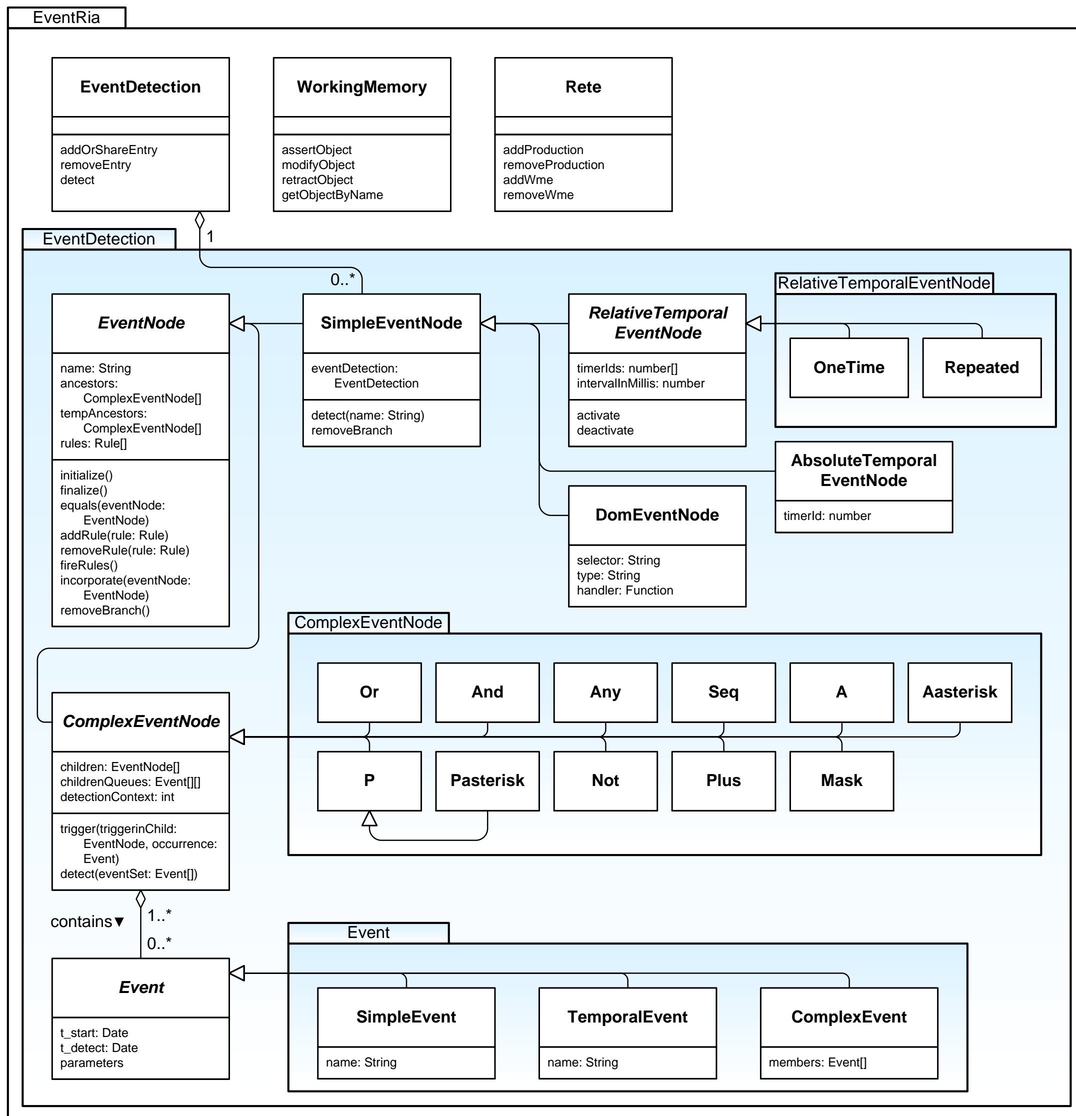

Figure 4.2: Event Detection (Class Diagram). This diagram shows the classes forming the event detection graph, along with classes representing the events propagated in the graph.

above, events can be deleted from input queues without side-effects on other event nodes, because each node keeps its own event queues for all input edges from child nodes.

Incoming events from child nodes are passed to the trigger method of a node. The method handles the deletion or queueing or detection of a complete pattern. In the latter case the new complex event is signalled to all parents. The constituent events are deleted if they cannot be part of further matches.

The actions of a trigger method are shown in Figure 4.3 exemplified by an event node `A` in Chronicle context. A child node has been triggered and detects a new event object of class `Event`, seen left-hand side in the figure. The child then triggers its parent, the aperiodic operator which has three children representing events of type E1, E2 and E3. The activities of the trigger method of `A` are shown in the right

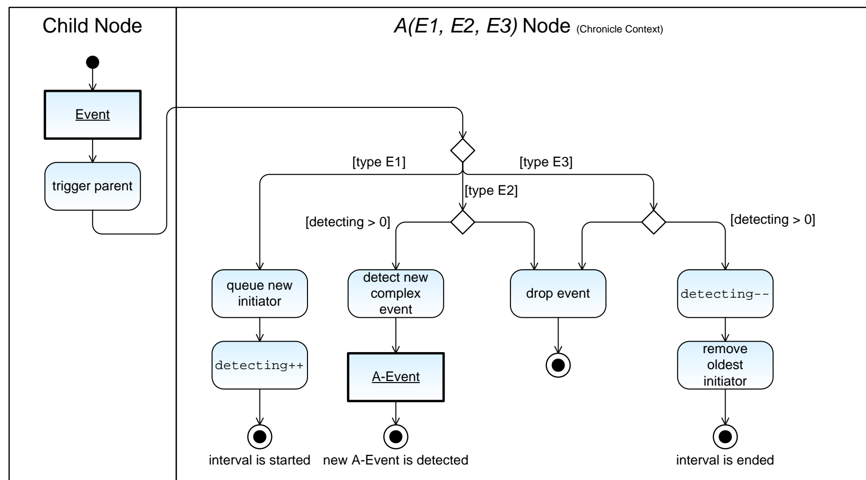

Figure 4.3: Triggering of the `A(E1, E2, E3)` event operator (Activity Diagram). This diagram shows the activities in an A event node, after it is triggered by one of its three child nodes. Depending on the type of the child node, the detected event is used as an initiator, an detector event or a terminator. Detector and terminator events are not used and discarded if no initiator has started a pattern matching.

"swimlane" of the diagram. Depending on the identity of the child, the supplied event is recognised for one of the positions E1, E2 or E3 of the event pattern.

The event operator `A(E1, E2, E3)` detects events of type E2 when they occur in an interval formed by occurrences of E1 and E3. In Chronicle context this means that occurrences of type E1 open an interval. They have the role of initiator events. Then each subsequent occurrence of E2 "detects" the complex event A. Events E2 have the role of detector events. If several initiators have opened intervals, then the detector is used for all of these. When eventually an event of type E3 is detected, it ends an interval. The oldest is selected of there are more than one. E3 has the role of terminator event.

In Figure 4.3 events E1 are therefore queued as initiators. The variable `detecting` which was initialised with zero is incremented. Depending on whether `detecting` is greater then zero, events E2 either cause a new complex event to be detected or are simply dropped. Events E3 end the detecting interval if any were opened. Otherwise events E3 are also dropped.

A new complex event is represented by a new event object of class `Event.Complex"-Event`. Event objects are used to make events explicit, so that they can be passed on, be stored, etc. A complex event object contains an array of its constituent events which led to its detection. The array is in the `members` attribute. Also, the complex event receives its two date attributes `t_start` and `t_detect` which are calculated to span the interval from the beginning of the oldest constituent event to the end of the newest constituent event.

For most event operators, the detector and terminator are the same. The example of the aperiodic operator is used above, because it is one of the more complicated

operators to compute. Additionally, the operator `A` demonstrates the difference between terminator events and detector events. For most operators the two are identical, for example for `And(E1, E2)`, the most recent constituent event on the one hand detects the complex event and on the other hand terminates the detection of the event, until another initiator occurs. The same holds for the sequence operator and most other operators. For the operator `Or(E1, E2)` only one constituent event of either type functions as initiator, detector and terminator fulfilling the pattern.

The complex event nodes for the different operators are subclasses of `Complex-EventNode`, see Figure 4.2. Except for their implementation of the trigger method, the node classes are identical. The method `detect(eventSet: Event[])` accepts a set of constituent events and creates the mentioned complex event object. The method then triggers all the parent nodes notifying them of a new event. The node classes contains arrays for links to the children and to store the input queues. Also, the user-selected detection context is stored in each node.

Nodes for simple and temporal events were not discussed, yet. The classes for these nodes are can be found in Figure 4.2, at the top of the package `EventDetection`, the area highlighted in blue. These nodes form the bottom of the detection graph, the entry points to the network. The nodes do not have any children, only parents.

The bottom of the graph is modelled as five hash maps in class `EventDetection`, left out of the diagram for brevity. There is one hash map for each of the simple event nodes: `SimpleEventNode`, `RelativeTemporalEventNode.OneTime`, `RelativeTem-poralEventNode.Repeated`, `AbsoluteTemporalEventNode` and `DomEventNode`.

The first of the hash maps is filled with the simple event nodes which are not in one of the sub-classes, which are discussed below. The keys in the first map are simple event names, the values are nodes. The hash map forms a sort of registry where every simple event is registered, and name clashes can be detected in advance by probing the hash map for the existence of a certain key. The event detection algorithm, lastly, uses the map, to distribute incoming simple event occurrences to the appropriate graph nodes.

The next hash map stores nodes of class `DomEventNode`. These nodes listen to certain DOM events. Such events are provided by the Web browser in which the Rich Internet Application (RIA) is running. The events include the basic events from Web programming like submit of a form, click of a button, etc. Most GUI programming in JavaScript is achieved through these events, cf. Section 5.1 on page 67 about JavaScript being single-threaded and the resulting approach of asynchronous programming. Handling DOM events is thus an important requirement for this work.

The remaining hash maps are used to store the temporal event types. Temporal events are are created by the system clock, which is invoked from the temporal nodes. Because the temporal events originate in their nodes, no distribution of events from external sources is needed. Still, the registry functionality of the hash maps is needed to keep track of all temporal event nodes in the system, and to

have references to them, when they must be cleaned up, for example on system shut down.

Temporal event types in Snoop are identified by a string representation, called *timeString*. Time strings are used as hash keys for temporal events instead of simple event names. Snoop defines time strings for relative and absolute time specifications. Relative time strings have a format like: "`6 mins 20 secs`", absolute time strings like: "`12:00:00/04/18/1996`". Relative time string can be used in a recurring fashion or just once, like a time-out. The recurring mode is needed for the periodic event operators `P` and `P*`. The simple time-out is used for the `Plus` operator. The two implementations are contained in the sub-classes `OneTime` and `Repeated`. Details of the implementations are discussed in the next chapter.

So far all node types are discussed. The event graph is built from these nodes. The graph grows by adding new sub-graphs which represent event expressions added to the system. This happens when new rules containing event parts are added to the rule engine. Rules without event parts obviously do not take part in event detection and therefore do not need to alter the event detection graph.

The event expression from a new rule is first converted to a separate graph of its own which is not connected to the event detection. The separate graph is constructed top-down from the nested event expression. The outermost event operator determines the class of event node which is used for the root of the separate graph. Then the process continues recursively for each sub-expression, creating children of the top node and attaching them. This is continued down to simple event nodes. Note that no sharing is done up to this point. The separate graph is a genuine tree, a bipartite graph with no two nodes connected by more than one paths.

When adding the new event expression, the separate graph is merged with the main graph. This is done from the bottom up, placing the simple event nodes from the separate graph in the hash map of the main graph or reusing existing equal nodes. The nodes which could be shared are added to a temporary list for further processing. Because these nodes had equal counterparts in the main graph, there is a possibility of yielding equality in their parents.

A pre-existing node is shared by using its reference instead of the newly created node. The latter is discarded in the process. Node sharing may only be utilised if it is equal to the new node. Equality is defined by the outcome of the method `equals` of class `EventNode` which is implemented by each graph node.

- Equality is given for two simple nodes when they have the same name.
- Equality is given for two complex event nodes if they are of the same class and use the same detection context and have identical children.

Checking for referential identity of children might seem like a conservative requirement, possibly ignoring further nodes which are equal. However, the event graph is

constructed bottom-up, so at any given state of adding nodes, the descendant nodes are already consolidated upon equality, so any equal nodes will be identical.

The described way of establishing equality for sub-trees works only when merging the event trees from the bottom up. The simple events at the bottom of the tree represent the literals of event expressions. The semantics of literals is axiomatic and indivisible. Therefore, it makes sense to build the semantics and with that equality of semantics upon these basic building blocks, in a bottom-up fashion.

As a side note, this is contrary to the construction of the Rete network which is discussed in Section 4.2 for matching rule conditions. The items for detection are inserted into the Rete network at a single root node. Therefore, new paths in the network are added top-down from the root.

To satisfy the requirement of deleting rules at run-time, the event graph must also be able to remove nodes after they have been added. The nodes in the graph therefore provide a set of methods to tear down nodes of a certain event expression.

The top node of each event expression is registered with its rule where e.g. the rule actions are stored. From this reference the method for removal can be invoked. The method is called `removeBranch()`. According to the node sharing status of its event node, the method determines whether to clean up its node and recurse down to the children. If the node is shared, the method returns, leaving the node unchanged. For temporal events this method also deregisters any timers with the system clock. For simple events their entry from the global hash map is removed.

This concludes the build-up and dismantling of the event graph. Nodes in the graph are shared to save space and prevent multiple computation of the same events. Sharing is respected when dismantling branches which are still needed.

4.2 Condition Matching

This section covers the design of the condition matching algorithm. According to the requirements collected in Chapter 3, a forward-chaining discrimination network is used. The Rete algorithm, cf. Section 2.5, is supplying the network. It has several similarities with the previously describes detection graph for events. Both are forward-chaining pattern matching algorithms. Both must be able to add and remove nodes at runtime, etc.

However, there are some important differences. Firstly it must be noted that they serve different purposes. In terms of semantics of rules [BrEc06], the event graph is concerned with transient, temporal data, i.e. events. The Rete network, on the other hand, is concerned with persistent data, representing the system state, i.e. business objects. The two types of data are to be separated in order to avoid making unnecessary events persistent, and thereby imposing a storage burden on an application.

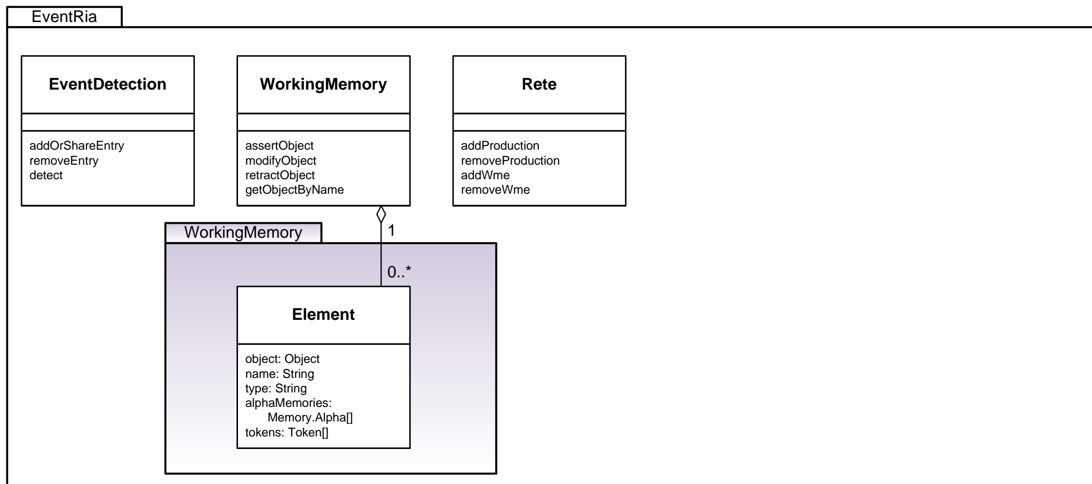

Figure 4.4: The Working Memory (Class Diagram). This diagram shows the working memory, a large hash map of working memory elements. The elements contain attributes `alphaMemories` and `tokens` which are used to store meta data from the pattern matching process in Rete.

The matching operations performed by the Rete network, use objects from the working memory to match the patterns specified in rule conditions. The working memory therefore describes the system state of a production system. Figure 4.4 shows the design of the working memory which is a large hash map of objects. For reasons of maintenance, the objects are wrapped in so-called working memory elements (WMEs), which contain an object along with additional maintenance information for the Rete algorithm.

Apart from the actual object and the maintenance information for Rete a WME also contains the attributes `name` and `type`. The attribute `name` contains a unique string identifying the WME. The string is defined by the user or rule when creating the WME. The string is used as the key in the hash map and is used as the identifier for retrieving the WME later.

The attribute `type` is also a string. It can contain a class name referring to an ontology. Specifying the `type` is optional when creating a WME, but it is needed if a user wants to retrieve the WME by its class later. The class can also be used in rule conditions by using an ontology comparator which can e.g. determine whether a WME belongs to a subclass of a given class. Further explanation of the operator is provided in Section 4.3 along with a simple ontology.

The class of an object is made explicit in a string attribute, as pointed out above. The mentioned ontology currently only supports a hierarchy. However, the classes are not modelled using the inheritance capabilities of the underlying programming language but instead by a string attribute. This is done because an inheritance hierarchy is limited to single inheritance. An ontology on the other hand offers extensibility through faceted classification. Therefore, the more flexible and extensible way of using the explicit attribute is chosen.

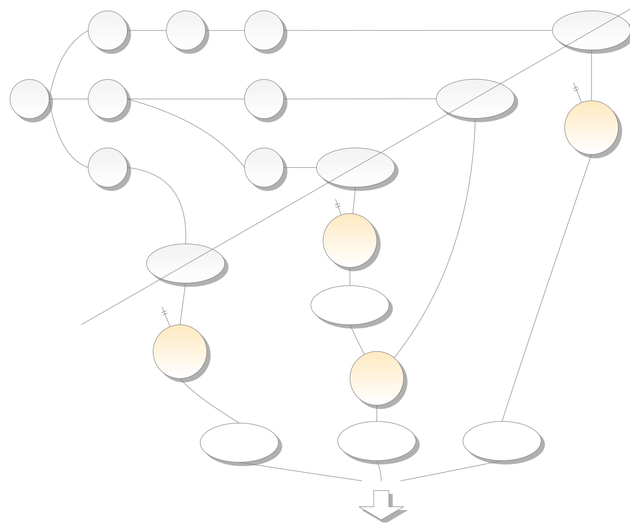

Figure 4.5: Rete Network Example. This diagram shows an exemplary Rete network. The alpha network is formed by *alpha nodes* in the top left area, each having one input edge. The alpha network ends in *alpha memories* which are located on the separation line of alpha and beta networks. The beta network consists of *beta nodes* which have two input edges, joining an alpha memory and a *beta memory.* The first beta nodes in a hierarchy of beta nodes are shown to be connected to a dummy input. The beta network ends in so-called *production nodes* which trigger their fully-matched rules at the bottom of the figure.

4.2.1 The Rete Network

Structurally, Rete is designed much more strictly than the event detection graph. The network for example is divided in two distinct parts which process input consecutively. Nodes have a fixed number of input edges, one input in the first part of the network, two in the second. The strict design stands in contrast to the event graph where nodes have varying numbers of children and there is no partitioning of the graph.

Conditions represented in a Rete network are formulae over objects. The formulae are patterns which must be matched with objects from the working memory. The network is created from these rule conditions which must to be evaluated to find the rules to run. Different checks and constraints from the rule conditions are transformed into a network of nodes. Each node of the network matches incoming objects to a part of the overall pattern and propagates any partial matches to subsequently find complete matches at the bottom of the network.

The patterns constrain single objects or declare variables across several objects. The latter requires the join of several objects to satisfy a suitable variable binding with more than one object from the working memory. Checks on single objects and joins of several objects are separated in Rete. They form two distinct parts parts of Rete, the so-called alpha network and the beta network.

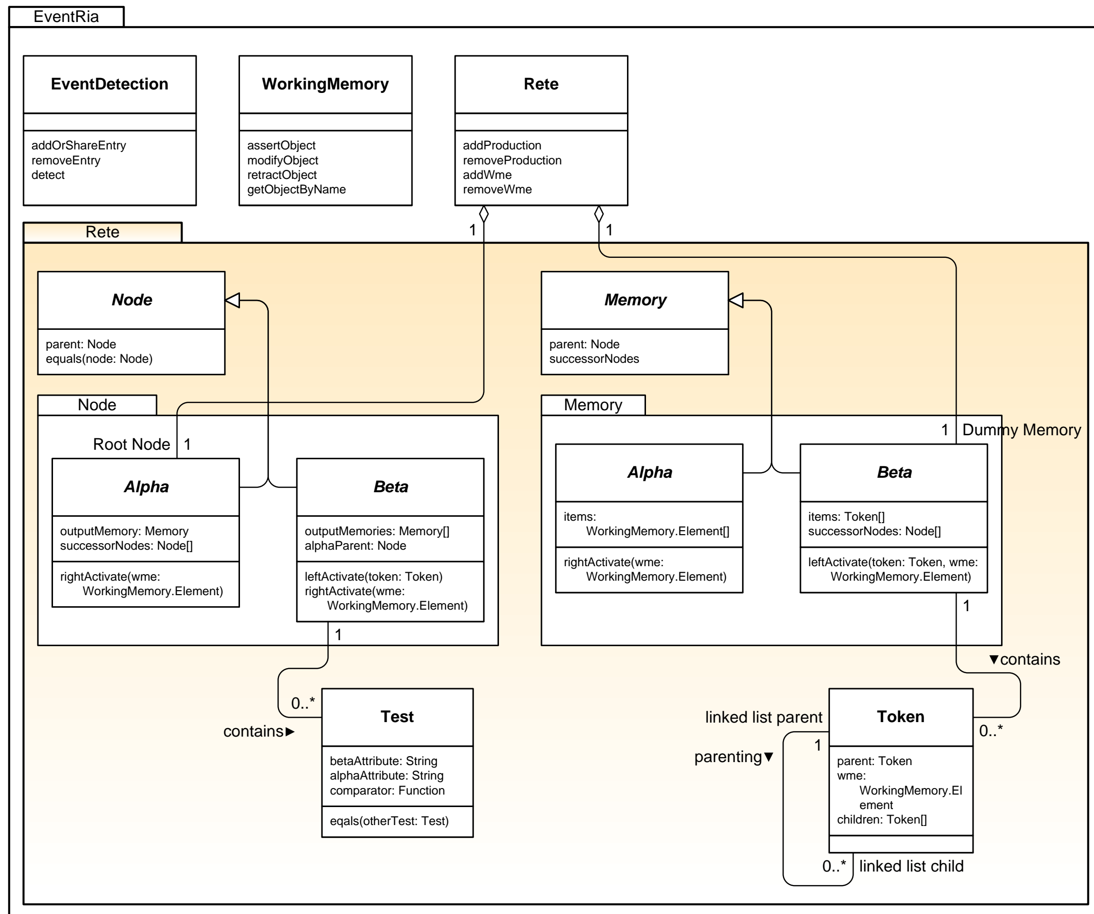

Figure 4.6: Rete Network (Class Diagram). This diagram shows the classes comprising the Rete network. The Rete class contains an alpha node in the role of the Root Node. Also, the dummy beta memory is connected to Rete. The rest of the network is reachable through objects of these two classes. Tokens are implemented as a linked list, so token objects are parenting token objects.

The alpha network contains *alpha nodes* and the beta network contains *beta nodes*. Additionally there are *alpha memories* and *beta memories* cf. Figure 4.6. Alpha memories are used at the end of a path of alpha nodes, forming leaves. The memories store objects which matched all nodes along the path and which can take part in joins in the beta network. Alpha memories, therefore, are the transitions between the alpha and beta network. Beta memories in the beta network are attached after each join node, i.e. beta node, to store the join results for further joining or to activate rules.

The alpha network forms a tree of nodes which have only one input edge for objects, cf. Figure 4.5. Each alpha node performs a check on each input object, and if an objects passes the check it is propagated onwards to all child nodes, which perform further checks. The leaves of the alpha network are formed by alpha memories which store any objects which are propagated down to them. Objects are stored in these memories to be used in joins in the beta network.

The beta network consists of nodes which perform checks across objects, i.e. inter-object checks, see Figure 4.5. This can be seen as joins in a relational database. Joins of more than two participating Rete memories are split into a series of two-input joins, each represented by a node. Each join node has one input edge from the beta network for previous joins and one input edge from the alpha network for the next memory. Each join node is followed by a beta memory, storing the join results. These results can either participate in further joins or represent complete joins which trigger the condition of a rule.

The data from unfinished joins represents partial matches of a complete rule condition. Only when a complete join is found the complete rule condition is fulfilled.

4.2.2 Adding Objects

Rete is used to determine whether objects match rule conditions. At run-time of the algorithm, objects are inserted into the alpha network. The objects from the working memory enter the tree at the root. They are inserted when they are new or have changed attributes, so that they must be re-evaluated by the Rete algorithm. The objects are then propagated along the alpha network, passing every node where the object fulfils the node condition.

When an object has traversed the alpha network, it has passed all conditions which are concerned with single object checks, i.e. intra-object checks. The object is then stored in an alpha memory.

The beta memory creates partial matches by joining one participating alpha memory after the other, until a complete join is found. Partial join data are the so-called tokens, which are lists of objects. These tokens are extended in each additional join. The tokens are stored in beta memories as join results. A join node following a beta memory tries to join each available token with each available object from the connected alpha memory, according to the join predicate from the rule. As mentioned above, a join node has two inputs, one beta memory supplying tokens and one alpha memory supplying plain objects.

The first join node in a series of joins starts with a dummy beta memory containing one empty token. This token is then joined with all objects from the adjacent alpha memory. For these join results, longer tokens are created, combining the dummy token and an object from the alpha memory. The longer tokens are stored in the attached beta memory. There they serve as input for further joins creating even longer tokens, built upon the first dummy token.

Tokens can be designed as arrays or as linked lists [Door95, Section 2.4]. The array design requires that for each join result, the participating token is copied to a new array with one additional field to accommodate the participating object. A token resulting from the join of N memories, occupies N units of memory. The child token from which the token was built also still resides in its respective beta memory. It is

one unit shorter, occupying $N-1$ units. Just as the child token of length $N-1$ is stored, are all its ancestors, down to the dummy token of length 0. This results in a space requirement of N factorial: $N!$ for array-form tokens.

Child tokens are not deleted as long as they are still valid partial matches. Keeping such intermediate results is part of Rete's approach to save state across matching attempts.

Designing tokens as linked lists provides some advantages and is hence used in this work. With list-form tokens, a join result is represented by a link to the parent token and a reference to the newly joined object. The hierarchy of links can be followed to traverse the complete token. No values are copied. This approach uses only 2 units of space for each child token, the complete hierarchy uses $2N$ units of space instead of $N!$. Additionally list-form tokens are instrumental in one of the object removal approaches described below.

4.2.3 Removing Objects

When objects in the working memory are deleted, Rete must delete all tokens containing the given objects. There are three fundamentally different methods of finding all tokens to an object, which are discussed in [Door95]. The methods are called rematch-based, scan-based and tree-based removal.

Rematch-based removal finds all affected tokens by inserting the objects mentioned above into Rete once more, but with a delete flag set. Every join result is then passed on with a delete flag and removed from the memories. This method of removal is simple and elegant because it reuses the existing traversal methods from adding objects for the removal as well. However, this method is slower than the removal methods described below. The cost of removing an object with rematch-based removal is equal to the cost of adding the object.

With scan-based removal an object which is to be deleted is also re-fed into the Rete network once more. However, the possibly expensive join conditions are not re-evaluated like with the previous removal method. Instead, the node which receives a deletion request sends it to its attached memory which scans its content for tokens containing the object. The affected tokens are deleted and the deletion request is passed on, to further child nodes. With this method the joins do not need to be re-computed and the scans also only affect memories which are in the path of the object to-be-deleted.

Tree-based removal is the most precise method of the three. It uses extra pointers which are created at the time of adding the objects. At the time of removal only those pointers must to be followed. Only the number of intended tokens must be visited, and no other computations have to be performed. The pointers which are recorded at the time of adding an object are listed as follows.

To be able to remove an object from the alpha network, all alpha memories are recorded which contain the object. This is done when adding the object to the

memory. Each object from the working memory is therefore wrapped in a working memory element. A WME contains the actual object along with additional maintenance information. One piece of information is an array containing the pointers to all alpha memories storing the given object.

The beta network, as pointed out above, groups objects from different alpha memories in lists, according to certain join conditions. These lists, i.e. tokens, must be revoked from the memories when one of their participating objects is removed. Otherwise a token, which represents a partial join result, may be part in further joins and yield results based on invalid join participants. Thus, every WME also carries an array of tokens it participates in. Upon removal of a WME these tokens are visited and removed. Because tokens are designed as linked lists, all child tokens which extend an invalidated token can easily be found and removed as well. This concludes the removal from the beta network.

When objects in the working memory are altered, the Rete network must also be notified. The implementation for this thesis handles modifications indirectly be removing and subsequently adding a WME to Rete.

4.3 Ontology

The client-side rule engine presented in this work allows limited access to an ontology. *Is-a* relationships can be determined using the class hierarchy from a Web Ontology Language (OWL) ontology. A server-side process is used to transform implicit knowledge from an ontology into explicit knowledge by reasoning. The so-called externalization can be done in advance on the server because the semantic annotations, e.g. class references, in client-side objects are known ahead of time, when Web pages are created.

The server-side process is a prototype written by Kay-Uwe Schmidt and is described in [SSST07, Section 5.1.3]. It converts OWL files to JavaScript Object Notation (JSON) in order to easily transport structured data to the client. The data contains a hash map of ontology classes. For each class there is an array holding its superclasses. Using this data structure, the client-side application can easily answer any queries for the *is-a* relation.

There is also a hash map for arbitrary relations, and the JSON format can easily be extended to make further features from the ontologies available on the client.

The *is-a* relationship is designed to be used as one of the binary operators in rule specifications. In rule conditions the operator can be used as a join condition for two classes. In event specifications the operator can be used to filter events by their parameters using the `Mask` event expression.

As a proof-of-concept a simple ontology is designed for the example of an algorithmic trading application. The classes of the ontology should conceptualise different

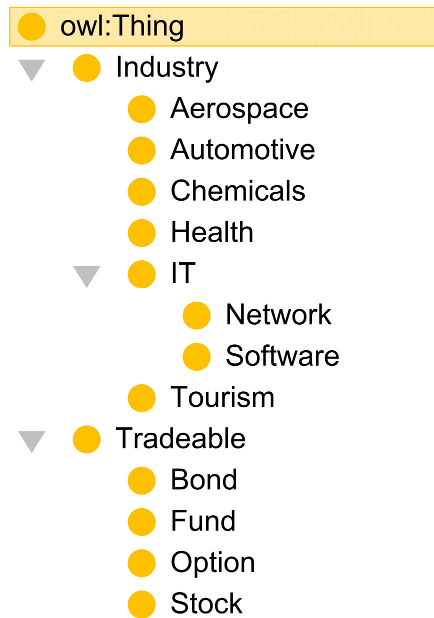

Figure 4.7: Tradeable Ontology (Asserted Hierarchy). This diagram shows classes from a minimalist ontology in OWL. The classes illustrate an ontology which can help decide whether a given tradeable thing, e.g. a stock, is from a certain industry in question. Answering such questions helps in filtering events or business objects according to a business vocabulary.

tradeable items like bond, fund, option and stock. The industries the items belong should also be conceptualised. For example, a stock and possibly a fund belong to a certain industry. The industries themselves should form a hierarchy to model larger industry sectors as well as specialisations thereof. This ontology can then e.g. be used to filter events from different stock price sources to only contain the IT industry.

Unfortunately no suitable existing ontology can be found as of the time of this writing. Two candidates are available which both were not applicable to this example.

The first available ontology is the *New York Stock Exchange (NYSE) listing ontology*[1] from the DAML Ontology Library[2]. The ontology contains some useful properties for stock instances. Yet there are no concepts of other, related tradeable items, and the industries were only conceptualised as a string *Datatype* property.

The second available ontology is the *Financial Ontology* [ABCG05] from the research project DIP[3]. Several properties are introduced for stocks along with other concepts for a stock market like Broker, Portfolio, BuySellOrder, etc. However, the industries are again only modelled as simple string properties.

[1]NYSE listing ontology. Online Resource. http://www.daml.org/2001/10/html/nyse-ont
[2]DAML Ontology Library. Online Resource. http://www.daml.org/ontologies/
[3]Data, Information, and Process Integration with Semantic Web Services: Project DIP. Online Resource. http://www.dip.semanticweb.org/

Therefore, an example ontology is designed from scratch using the OWL editor Protégé[4]. The classes of the new ontology are shown in Figure 4.7. The ontology conceptualises tradeable items: bond, fund, option and stock. Also conceptualised are the industries the items belong to. For example, a stock and possibly a fund belong to a certain industry. A relation *fromIndustry* is used for this purpose.

The class hierarchy from the ontology can be queried to form filter expressions in the event part of rules, as well as object patterns in the condition part of rules. The event filter expression is the `Mask` operator, described in Subsection 4.1.3 as the design of a content-based check on events. For object patterns in conditions, can use an ontology query as a join predicate, as described in Subsection 4.4.2.

4.4 Client-side Rule Language

The rule language for this work supplies reaction rules to a rule-based application. Reaction rules are triples of event-condition-action. The event specification from a rule is used to extend the event graph, whereas the condition specification is used to extend the Rete network respectively. Actions are stored in a data structure which is accessible from both the event detection and condition matching structures.

From a user's point of view the rule language is the interface to programming and adapting the rule-based Internet application. Figure 4.9 on page 64 shows the user's interaction with the RIA. Interaction is communicated through rule specifications which the user passes via an adapter to the Graph Builder. The Graph Builder subsequently interacts with the event graph, Rete network and rule repositories according to the specifications from the rules.

For the rule language the JavaScript-friendly JSON format is chosen. JSON is published as a Request for Comments (RFC) [Croc06]. Like XML it provides a structured representation of data with deep nesting. Unlike XML it is readily usable in JavaScript because JSON syntax is the subset of JavaScript otherwise used to denote objects literals and array literals in the programming language. Although JSON is JavaScript there is a thin parsing layer involved to provide security from introducing executable code. Other than that, JSON uses a very lean syntax compared to XML. Tags do not need to be named if, for example, they are just used to provide structure like nesting. JSON uses curly brackets to delimit nested expressions. Thus, a lot of markup can be saved.

JSON can be used to maintain nested data, therefore the rule language for this work can be formulated in JSON as an abstract syntax tree. A similar approach is taken by many modern XML-based languages, like RuleML and its ECA rule standard, Reaction RuleML [PaKB07].

Using an abstract syntax tree to transport the language relieves the client-side application of parsing any expressions. Instead the nesting of expressions can be easily

[4]Protégé ontology editor. http://protege.stanford.edu

determined by descending the supplied tree. Also, no aspects of concrete syntax must be retained when abstract syntax is used.

The design of the language is explained along with some examples. The complete grammar is designed in (extended) Backus-Naur Form (BNF). It is shown in full in appendix Chapter A. The given BNF contains minor additions for the parser generator tool ANTLR[5]. The tool was used to edit and later test the designed grammar.

The grammar describes a so-called rule file. The rule file is the granularity at which rules are transported, e.g. downloaded into the rule framework. A rule file may contain more than one event-condition-action (ECA) rule contained in a rule set. Meta data for the rule set are also part of the rule file and a library of reusable event expressions.

As mentioned above, JSON is used as an underlying format for the proposed rule language. The language therefore is a specialisation or a subset of JSON.

The syntax of JSON can describe strings, numbers, the boolean literals *true* and *false* as well as objects and arrays. Objects are enclosed in curly braces. They contain a comma separated list of attributes. An attribute is a string followed by a colon and the value. The value might in turn be any JSON expression. Arrays are enclosed in square brackets, containing a comma separated list of expressions.

The proposed language restricts tree-expressions from JSON in way that only certain objects with certain attributes may be used and nested. The language is therefore a subset of JSON which is defined by a BNF grammar.

An example rule file in JSON is depicted in Listing 4.1. It consists of an object in curly braces starting at line 1. It contains an attribute `"meta"` as a meta data section to store information like an identifier for the rule set, an author and version information as strings or numbers. Optionally the meta data section may be extended by further name-value assignments for future enhancements to the rule engine.

The next attribute `"eventBase"` may contain predefined events by the user, which can be referenced by name from event specifications in the rule set. Rules, lastly, are listed in the `"rules"` attribute of the rule file at line 8.

```
1  {
2      "meta": {
3          "ruleSet": "RuleSet1",
4          "author": "Roland Stuehmer",
5          "version": 1.0
6      },
7      "eventBase": {},
8      "rules": [
9          ...,
10         ...,
11         ...
```

[5]ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/

```
12     ]
13 }
```

Listing 4.1: Syntax Example of a Rule File

In the grammar the top production for a rule file is the following, with line numbers referring to the complete grammar in appendix Chapter A:

```
7 ruleFile : '{' (ruleSetMetaData ',')? (eventBase ',')?
8     (conditionBase ',')? (actionBase ',')? RULES ruleSet '}';
9
```

The grammar rule contains an optional meta data section, an optional event base, optional condition base and optional action base. Then follows the "rules" attribute with an array forming the actual rule set.

The condition and action base are thought of as repositories of reusable components, however, unlike the event base, they are not part of the implementation of this work. These two parts of the rule language are left blank. The condition base may be used in further work to specify reusable condition parts and the action base may be used to supply a predefined set of rule actions according to a vocabulary of actions like *user interface patterns* from [Tidw06]. So far this covers the structure of the rule file.

An example of a single rule is depicted in Listing 4.2. The rule is formed by another object in culy braces, containing three attributes for event, condition and action specifications.

```
1 {
2     "event": {
3         ...
4     },
5     "condition": [
6         ...
7     ],
8     "action": [...]
9 }
```

Listing 4.2: Syntax Example of a single Rule

The event attribute is an object containing either an event or a complex event operator. In the latter case the event operator contains further events, nested as children.

The condition attribute consists of an array. If more than one array element is specified, the elements form a conjunctive condition. This behaviour is adopted from the Drools rule language [PNFJ+08]. It allows the user to easily specify a list of conditions which all have to be fulfilled.

The action attribute also forms an array. This is needed because several actions of different types might be required for a fulfilled rule. This satisfies the requirements from Section 3.3 to offer imperative programming as well as declarative programming and firing of events from rule actions.

Optionally a rule may contain an attribute `"meta"` which stores a rule name, author name and version information on a per-rule level. If present, this data is used by the rule engine for error reporting and logging information.

The BNF grammar for a rule including the optional meta data part is:

```
22 reactionRule : '{' (ruleMetaData ',')? (eventPart ',')?
23          (conditionPart ',')? actionPart '}';
24
```

A single reaction rule contains an optional meta data section, followed by an event part and a condition part. The event part may be missing for CA rules and the condition part may be missing for EA rules, cf. Section 3.2. The concluding action part is mandatory in all cases.

4.4.1 Event Part

A rule in the proposed rule language may contain an event part. It is used to specify event patterns using Snoop operators, and additional operators from the requirements analysis in Section 3.1. Namely these additional operators are the event mask and the DOM events. It is a requirement for the rule language to exhibit all operators to the user.

It must be noted that the event part is optional for the case of CA rules. However, for reactive rules like EA or ECA rules, the event specification is an integral part of the definition, determining when a rule is fired.

The event part of a rule consists either of a simple event or of a complex event specification. A complex event specification is an event operator, for example from the event language Snoop, which has several sub-expressions, like the operator *And(E1, E2)* with two child expressions.

The different operators are distinguished by the value of the `"type"` attribute. Further attributes are determined by that value. For an aperiodic event, `"type": "A"`, a declaration of children events is expected, and optionally the detection context. For detection contexts, cf. Subsection 2.1.2.

Child expressions in the rule language are modelled as elements of an array called `children[]` which is part of each complex event type. The elements of the array are in turn arbitrary event expressions, separated by commas.

Listing 4.3 shows an example of an aperiodic event specification. The event type is specified, followed by three children and followed by the detection context. Specifying the detection context is optional and Chronicle is the default value if the specification is left out. For the children, three events are given, determined only by their name.

```
1 {
2     "type": "A",
3     "children": [
4         {"name": "a"},
5         {"name": "b"},
6         {"name": "c"}
7     ],
8     "context": "CHRONICLE"
9 }
```

Listing 4.3: Syntax for an aperiodic Event A()

These names are first searched in the event base. If the rule author has specified matching event expressions there, then they are reused at this point and are inserted when adding the rule to the rule engine.

If no entries of the same name are found in the event base, the name is treated as a simple event. It then forms a leaf node of the event graph.

The grammar for the event part of a rule is designed as follows:

```
36 // Event
37 event
38     :   '{' (eventExpression | eventName) '}';
39         // Either an elaborated complex event expression or
40         // the name of a predefined or simple event.
41         // Predifined events are specified in the eventBase.
42
43 eventExpression
44     :   eventTypeExpression
45         // Used with the ANY(m, E1,...En) event:
46         (',' M NumericLiteral)?
47         // Used for everything except TEMPORAL and DOM:
48         (',' eventChildrenExpression)?
49         // Used with P, P*, PLUS and TEMPORAL events:
50         (',' timeStringExpression)?
51         // Used with MASK event:
52         (',' maskExpression)?
53         // Used with DOM event:
54         (',' domExpression)?
55         // Used with P and P* events:
56         (',' paramsExpression)?
```

```
57        // Event detection context:
58        (',' contextExpression)?;
59
```

An event, on line number 38 of the grammar, is either a complex event expression, containing a type attribute and others, or the event is simply an event name, as shown in the example of the events named “a”, “b” and “c”.

A complex event expression like in the example of the aperiodic event “A” is concretised on line 43 of the grammar. It must contain a type attribute. The remaining attributes depend on the type of event, or are altogether optional, like the detection context on line 58.

The event `Any(m, E1, ..., En)` from Snoop must contain the extra attribute `"M"` to store the number of child events to occur before the Any event occurs, cf. line 46.

The `eventChildrenExpression` is required for all complex events relying on input from less complex events. Only temporal events and events from the DOM do not do so, cf. line 48.

Temporal events, on the other hand, require a `"timeString"` attribute. It specifies, when an event, driven by the system clock, will occur. The P, P* and Plus events also use a timeString attribute, to specify relative time strings as an offset of time, cf. line 50.

The `maskExpression` is used with the `Mask` event type. That event type specifies either a predefined masking function or the JavaScript code for a user-defined function. The array `argumentsDefinition` is either passed to the specified predefined function or in case of a user-defined function, the first argument contains the actual function. The grammar for the `maskExpression` is this:

```
66 maskExpression
67    :   MASK '{'
68            // Optional when maskType is "JAVASCRIPT".
69            (maskTypeExpression ',')?
70            argumentsDefinition
71        '}';
72
```

The next event attribute is the `domExpression`, cf. line 54, which is also comprised of two parts, like the mask expression. The DOM expression is meant to specify events from user interaction with the Document Object Model of the Web page, i.e. RIA. A DOM event is specified by a pair of selector and event. A so-called selector specifies a certain DOM node or a set thereof where events may be observed. The event specifies the type of DOM event which is observed. Possible types are “click”, “dblclick”, “keypress” and more. These events are raised by the Web browser and

must be handled by rule actions like any other event. The following excerpt from the grammar accommodates the selector and event for a DOM event specification:

```
73 domExpression
74     :   SELECTOR QUOTEDSTRING ','
75         EVENT QUOTEDSTRING;
76
```

The next attribute of an event specification is the parameter specification contained in `paramsExpression`, cf. line 56. This expression is required for the periodic events P and P* from Snoop. These event types occur periodically at a user-defined interval specified in the `timeString` attribute.

The last event attribute is the context specification. As mentioned before, it is optional. The default context, Chronicle, is assumed if there is no specification. The context determines the selection of individual events if more than one is queued by a child event operator. Simple events do not have children, therefore a context specification does not apply to them.

This concludes the list of attributes which can be used in an event specification.

4.4.2 Condition Part

Additionally to an event part, a rule may contain a condition part. The condition part specifies patterns which are matched against objects from the working memory, as opposed to stream data in the case of events.

For the design of the proposed rule language, object patterns were compared from different rule languages. A short example of a pattern in different languages is given in the following. The example pattern matches a person with a certain cheese the person likes. It is taken from the Drools documentation [PNFJ+08].

A rule language offering succinct patterns of objects including their attributes is the language from the project DEVICE, [BaVl97]. Another language with a non-verbose syntax is Drools [PNFJ+08]. Both are compared to the syntax of the XML-based Reaction RuleML [PaKB07].

The rule example displaying syntax from *DEVICE* is shown in Listing 4.4. The example shall demonstrate a rule employing a join. The condition following the `IF` keyword joins an object of class Cheese on line 1, with an object of class Person on line 2. The matching instance of Person is later accessible through the variable `pers1`, declared before the class, and the “@” sign.

Further constraints for the matching objects are declared in parentheses after the the class specification. The constraints may also declare variables which must be consistent for a completely matching set of objects.

A matching object of class Cheese must have an attribute “name” with a value equal to the string “cheddar”. Also, the value of the attribute is declared to bind to the variable X.

A matching object of class Person must have an attribute “favouriteCheese” with an value matching any prior bindings of the variable X, i.e. the person’s favourite cheese must also contain the string “cheddar” is this example.

```
1 IF   Cheese( name="cheddar", name:X ) and
2      pers1@Person( favouriteCheese=X )
3 THEN do-something-on-pers1
```

Listing 4.4: Example condition syntax from DEVICE

The syntax from *Drools* is very similar and also quite succinct. Drools also follows the overall syntax of specifying a class restriction, followed by parentheses with further attribute restrictions. Consecutive lines in Drools are assumed to form a conjunction.

Variable bindings for matched objects are also supported, separated by a colon from the class name. Drools encourages its rule editors to use variable identifiers starting with the “$” sign to make them easily distinguishable.

```
1 rule
2 when
3     Cheese( $chedddar : name == "cheddar" )
4     $person : Person( favouriteCheese == $cheddar )
5 then
6     do-something-on-$person
7 end
```

Listing 4.5: Example condition syntax from Drools

An example of *Reaction RuleML* is shown in Listing 4.6. This language uses an XML representation and thus forms a tree of expressions. Syntax elements like “<Atom>” are made explicit. No parsing is required to generate a tree structure, except for generic XML parsing. However, the language is quite verbose.

```
1 <if>
2 <And>
3 <Atom>
4     <op>
5         <Rel>name</Rel>
6     </op>
7     <Var type="Cheese">C1</Var>
8     <Ind>cheddar</Ind>
9 </Atom>
10 <Atom>
11     <op>
12         <Rel>favouriteCheese</Rel>
13     </op>
14     <Var type="Person">P1</Var>
15     <Var type="Cheese">C1</Var>
```

```
16 </Atom>
17 </And>
18 </if>
19 <do>
20     do-something...
21 </do>
```

Listing 4.6: Example condition syntax from Reaction RuleML

The proposed rule language from this thesis also uses a hierarchical data representation like XML, namely the JSON format. However, it is less verbose than Reaction RuleML.

Parts of the verbosity of Reaction RuleML is due to the underlying language RuleML. RuleML is designed to support different logic systems and must conform to several rule standards and systems. Description logic, Horn logic and Prolog are mentioned as interchangeable foundations to formulating rules supported by RuleML [Bole01].

The proposed rule language uses condition specifications inspired by the Drools language. Some structure is introduced in the language to relieve the interpreter of any lexical analysis or parsing, after the JSON parsing. The JSON parsing is generic like XML parsing and different implementations are readily available. A discussion of JSON libraries will be part of Section 5.2 in the chapter on Implementation.

```
1  "condition": [
2      {
3          "class": "Cheese",
4          "fields": [
5              {"field": "name", "comparator": "==",
6                  "literal": "cheddar", "vardef": "$X"}
7          ]
8      },
9      {
10         "class": "Person",
11         "vardef": "$person",
12         "fields": [
13             {"field": "favouriteCheese", "comparator": ">=",
14                 "variable": "$X"}
15         ]
16     }
17 ]
```

Listing 4.7: Example condition syntax from JSON-Rules

Listing 4.7 shows the previously used example in the syntax of the proposed rule language. The rule condition starts at line 1. It represents the attribute "condition" of a rule containing attributes event, condition and action. The condition forms an array of conjunctive class patterns. These patterns may bind variables like in Drools.

The first array element of the conditions array starts at line 2, the second element at line 9. Each element contains a "class" declaration. The declared class restricts any matching objects to belong to this indicated class.

The class declaration remains optional as a concession to programming in JavaScript in a dynamical and flexible manner. This satisfies the requirement of honouring JavaScript programming practices from Section 3.4. When the class declaration is left out, every object from the working memory may satisfy the remaining constraints, regardless of its class.

The second condition element contains a “vardef” declaration on line 11. Specified on the object level such a variable declaration binds any matched object to a variable name which will be accessible for the rule actions. There is also a “vardef” declaration on the attribute level within objects which is explained in the next paragraph along with other attribute level operations.

Both condition elements in the example contain an attribute “fields”, which contains a series of field constraints for objects to match the pattern. Field constraints are imposed on the attributes, i.e. fields, of a matching object. In the above example there is only one field constraint per object. The constraint for the first matching object, starting on line 5, restricts the object to having a field called “name” which must satisfy the comparator “==” with the literal string “cheddar”. Additionally the matching field is bound to the variable “$X” in the “vardef” declaration on the field level.

The second object also has only one field constraint, starting on line 13 of Listing 4.7. In this constraint the given field, “favouriteCheese”, must match a variable, not a literal. The specified variable is “$X” which has previously been bound to the “name” field of the first object. This results in a join of the two objects according to the join predicate using the specified “>=” comparator.

The grammar of field constraints is given below, from line number 120 onwards. For the right hand side of any comparator, there are several possible ways of accessing a value. The previous example used “literal” for a direct operand and “variable” for the value of a previously bound variable. Additionally any field may also be compared to a second field of the same object, facilitating intra-object constraints. This is specified in the rule language by “field2” instead of “literal” and “variable”.

```
101 // Condition
102 conditionElement
103     :   '{'
104         // either the class and name may be left out...:
105         ((
106             (CLASS QUOTEDSTRING ',')?
107             (NAME QUOTEDSTRING ',')?
108             (VARDEF QUOTEDSTRING ',')?
109             FIELDS '[' (conditionField
110                 (',' conditionField)*)? ']'
111         ) |
112         // ... or the fields may be left out:
113         (
```

```
114          (CLASS QUOTEDSTRING | NAME QUOTEDSTRING |
115             CLASS QUOTEDSTRING ',' NAME QUOTEDSTRING)
116          (',' VARDEF QUOTEDSTRING)?
117       ))
118       '}';
119
120 conditionField
121    :  '{'
122       FIELD QUOTEDSTRING
123       (',' COMPARATOR QUOTEDSTRING (
124          (',' LITERAL flatLiteral) |
125          (',' VARIABLE QUOTEDSTRING) |
126          (',' FIELD2 QUOTEDSTRING))
127       )?
128       (',' VARDEF QUOTEDSTRING)?
129       '}';
130
```

For comparators the usual binary operators are available: `<, >, <=, >=, ==, !=`. These are complemented by set operators like: “contains” and “notContains” as well as an ontology query operator and some JavaScript specific operators. The latter type of operators provide some proximity to JavaScript for Internet programmers, in order to satisfy the requirement for honouring JavaScript programming practices from Table 3.4.

The set of available operators is organized in a hash map. Each operator has a string identifier like “<” or “contains” used in the rules, cf. line 123 of the above grammar, and line 5 and 13 of Listing 4.7. The mentioned string identifier is also used as the key of the hash map, mapping the identifier to its suitable implementation. By using a global hash map in the rule framework to store the available operators, a single point of entry is created to easily extend the set of operators offered by the framework. Any binary function with boolean output can be used as an operator.

The ontology query operator is also a binary function returning true or false. The operator is named `onto:IS_A` in the rule language and thus in the hash map of operators. It accepts two class names and checks the prepared ontology on whether the first class is a sub-class of the second. Using this operator allows for rules e.g. checking whether some field of an object is a sub-class of some given class.

In the example of an algorithmic trading application it can be determined whether a stock is an IT stock by considering a possible `fromIndustry` field. Using a suitable ontology it can be decided that a value of “Software” represents a sub-class of “IT” as well as “Industry”. The operator `onto:IS_A` is therefore used like any other implementation of a constraint or join operator.

4.4.3 Action Part

Every rule in the proposed rule language must have an action part. To meet the requirements from Section 3.3, three types of actions should be provided.

The first type of action, which is a mandatory requirement, contains arbitrary JavaScript code. The action type to accommodate this is specified by “JAVASCRIPT” in a rule action. The action type is represented on line 133 of the grammar below. Even though this approach is not a way of declarative programming when the actions themselves are concerned, this facilitates the greatest flexibility and extensibility for rule-based programming. It is therefore one of the requirements.

```
131 // Action
132 action : '{'
133         TYPE QUOTEDSTRING  // '"JAVASCRIPT"' | '"EVENT"'  | '"ASSERT"' ...
134         (',' FUNCTION QUOTEDSTRING)? // Used with JAVASCRIPT
135         (',' TRIGGER QUOTEDSTRING)? // ... with EVENT
136         (',' CLASS QUOTEDSTRING)? // ... with ASSERT, optionally
137         (',' NAME QUOTEDSTRING)? // ... with ASSERT, RETRACT and MODIFY
138         (',' OBJECT literal)? // ... with ASSERT
139         (',' MODIFY QUOTEDSTRING)? // ... with MODIFY
140         '}'
141         ;
```

The traditional, declarative way of providing state-changing action in a rule system is to declare changes of facts. To that end usually a small set of operators are offered, like *assert*, *retract* and optionally *modify*. Rule engines called production systems using this approach were previously mentioned. Examples are OPS5 [BFKM85] and CLIPS [CuRD93].

For an object-oriented rule engine such facts are represented as objects. They are contained in a working memory as they are for the traditional rule engines. The three operations are applicable for objects in the following manner. Assert adds a new object to the working memory, creating a WME. Retract deletes an object i.e. its WME from the working memory. Lastly, modify applies a specified function to the object. All operations notify the Rete network to re-evaluate the affected object.

For the proposed rule language these operations form the second type of rule actions. With an action type of “ASSERT” an rule author may specify the class of the asserted object, cf. line 136 of the grammar, the name of the asserted object, cf. line 137, and finally the actual object, cf. line 138. The object may be any JSON literal. Basically this includes arrays, objects, strings, etc, and arbitrary nesting thereof. For specifying complex nested data types in JSON, see the first paragraphs of Section 4.4.

With an action type of “RETRACT” a rule author specifies the name of the WME to delete. However, with a type of “MODIFY” the user also specifies a JavaScript function to conduct the desired modifications.

The third type of action is the explicit event, an event which is triggered as the consequence of a rule. The rule language provides an easy way to specify such an action as follows. The action type must be “EVENT” and the “trigger” attribute must be the name of the simple event to trigger, cf. line 135 of the grammar. This concludes the three types of rule actions.

4.5 Overall Architecture

The overall architecture of the proposed rule framework is shown first on a class level and then on a higher level component view which additionally shows some peripheral agents.

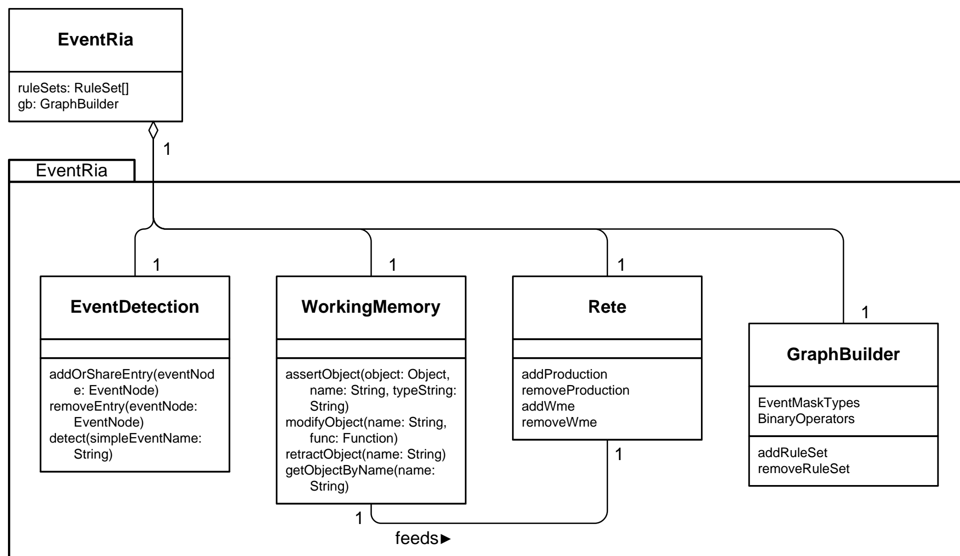

Figure 4.8: EventRIA Main Class (Class Diagram). This diagram shows the main classes of the framework: EventDetection, WorkingMemory and Rete. The GraphBuilder contains methods to uniformly access the other components.

The architecture as seen on a class level is shown in Figure 4.8. The important classes described in this chapter are combined here by association to an overarching class `EventRia`. EventRia encapsulates the complete framework. It contains the different parts of the engine and allows for a uniform access to its functionality.

There is one event detection graph, one working memory and one Rete network associated to the EventRia class. The working memory is shown to feed the Rete network with updated WMEs. The class `GraphBuilder` contains methods to accept rules and thereby adapt the engine. The graph builder distributes the appropriate parts of rules to the event detection, the Rete network and the rule repository, respectively, which is holding the rule actions.

Figure 4.9 shows the event-driven rule framework from a more abstract perspective, using a component diagram. Event sources on the left of the figure feed different

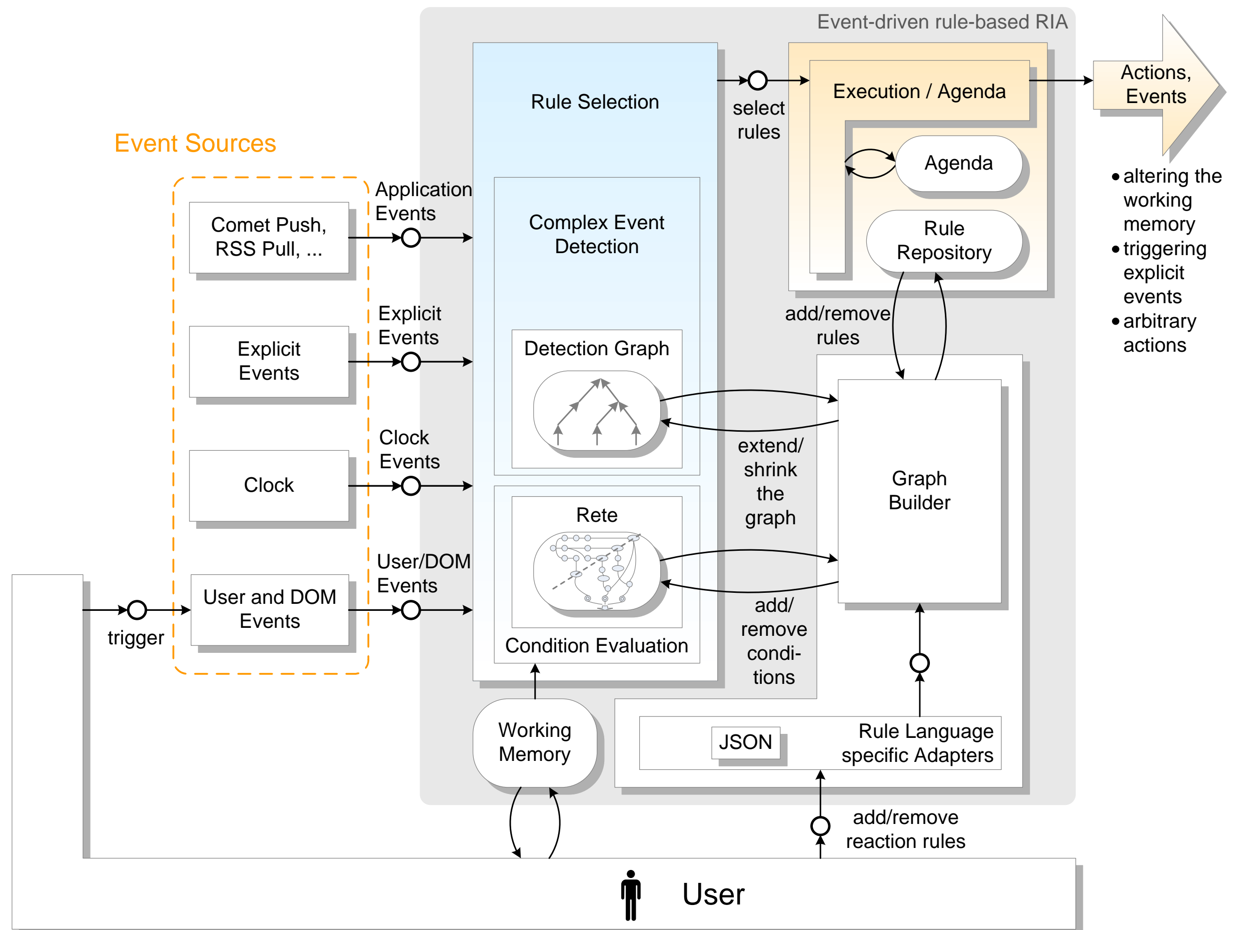

Figure 4.9: System Overview (Component Block Diagram). This diagram shows the overall architecture. Event sources send events to the rule selection. The rule selection detects complex events from these and finds fulfilled conditions from the working memory. Then rules are selected and executed. The user may trigger events, alter the working memory and customize the application by adding reaction rules through the graph builder to the appropriate parts of the system.

types of events to the rule selection in the middle. The rule selection determines the rules to run on the basis of detected complex events and verified working memory patterns. The rules are then fired, entailing different reactions, depicted in the arrow on the right.

The user may add and remove rules, on the bottom right of the figure, adapting the logic of the application. The user also modifies the working memory, bottom centre of the figure, changing the state of the application. Thirdly the user triggers events, bottom left of the figure, providing one of the sources of temporal data for a reactive application.

Component diagrams contain two different entities. Active entities are depicted by rectangular shapes. Passive entities are depicted by rounded shapes. Active entities, or *agents*, may represent either artificial agents, the default, or human agents marked

with an according pictogram like the user in Figure 4.9. Passive entities for example resemble data *storage* in an application.

Agents may read and write storage components, according to the arrow directions. Modification arrows contain both directions and are depicted as curved arrows, e.g. between the user and the working memory. To enable agents to communicate directly with each other, so-called *channel connectors* are used which are arrows with a small circular storage in the middle. The direction of data flow can be indicated by arrows. Channel connectors are used e.g. between each event source and the rule selection component.

When components are nested, the outer component implicitly has access to the inner component. Therefore, no extra connectors are needed for example in the rule selection and its enclosed components. The contained event graph and the Rete network are part of the enclosing components. This concludes a brief overview of component diagrams.

The event source *"Comet Push, RSS Pull, . . . "* needs further explanation. Different origins of events are discussed in several places of this thesis, for example in Section 3.1. The user, the system clock and remote Internet applications are named as possible sources for events. When subscribing to events, it must be clarified, how the events reach the subscriber. There are pull strategies and push strategies for Internet applications. Push strategies have several advantages. Latency is reduced because an event is dispatched to the subscriber as soon as it occurs. Load is reduced on both the subscriber and the publisher because there is no unnecessary polling.

To use a push strategy for transporting events to a RIA, this thesis uses the Comet architecture introduced by Alex Russell [Russ06]. Comet is a set of methods to keep a long-lived Internet connection open between a Web server and client. Using such a connection, a server can send event data to the client without waiting for a periodic request or the like. A third-party Comet implementation is used for this thesis to test the example application of algorithmic trading which receives simulated stock price changes from a testing Comet server.

Apart from the components with the core functionality like Rete and event detection, other components are part of the design. Several adapters are to be implemented in different components of the proposed framework. The incoming events from the Comet server must be manifested into first class objects in order to propagate and store them in the detection graph. This is done by an adapter. The latter creates new objects for the events, adds the time stamp when it is received and preserves any other data like parameters. The event is then fed into the detection graph. Adapters for other event sources may be added, e.g. to facilitate the polling of RSS feeds, and the construction of event objects for the feed items or for detected changes in repetitive feed items.

The graph builder component is another adapter. It converts the declarative rule language into internal data structures like event nodes and Rete nodes. The adapter dissects the rules. For the event part the rule expressions are turned into nodes for

the detection graph. The graph builder incorporates them into the graph, reusing common sub-graphs that it can detect among the newly added nodes and the existing graph. Among the detected similarities are identical sub-graphs, identical temporal events and identical simple events. Chapter 5 elaborates on establishing identity between nodes and entire sub-graphs. For the condition part of a rule the adapter creates the Rete network to recursively evaluate a condition along the nested operator nodes as described previously. For the action part of rules the adapter maintains the rule repository.

5

Implementation

The preceding chapter focused on the design of the proposed framework. The framework consists of a few major components to achieve event-driven reactivity and adaptability. This chapter now illustrates the creation of the framework on a more language specific level. At first an introduction to the language JavaScript is given, then the components of the framework are described one by one, and after that their integration into a single framework is pointed out. Ontologies are a sort of cross-cutting concern for this chapter, as they are required in more than one component of the framework. Therefore, ontologies will be mentioned several times during the following sections. The chapter starts out with a few facts about the programming language JavaScript. These might be interesting to a reader with a background of less dynamic programming languages like Java.

5.1 JavaScript Language Features

JavaScript is the widely used term for *ECMAScript*, standardised[1] as ECMA-262 by Ecma International, formerly the European Computer Manufacturers Association. The term JavaScript originates in a dialect of the language which pre-dates and heavily influenced the standard. This work shall refer to ECMA-262 as *JavaScript*.

Contrary to what its name suggests, JavaScript has not much in common with the language Java. Both borrow parts of their syntax from C and C++. Otherwise,

[1] ECMAScript Language Specification. Online Resource. http://www.ecma-international.org/publications/standards/Ecma-262.htm

however, the languages are very different. JavaScript supports dynamic typing, its functions are first-class objects, and its inheritance is prototype-based. These characteristics and some more are elaborated briefly as follows.

dynamic typing: JavaScript allows variables to change their types at run-time. More precisely this means that values have a type and when a value is assigned to a variable, the variable changes its type to the one from the right-hand side of the assignment.

first-class functions: Functions in JavaScript are objects. This allows for higher-order functions, so passing and returning functions is possible. Functions may be invoked with the `new` keyword, returning a new object, initialized with the inner variables of the function. Thus, functions can be used as constructors.

closures: Inner functions may retain bindings of variables bound by an outer function, even after the outer function has returned. Thereby, a *closure* is formed around the inner function. The values stay bound with the inner function when it is handed on or is invoked.

prototype-based inheritance: JavaScript possesses no classes. Objects may inherit from objects, preserving a link to the parent in a "prototype" property. In an inheritance hierarchy the chain of prototypes is followed by the run-time environment to find functions implemented by parent objects. The so-called prototype chain is also used to verify whether an object is the descendant of another given object.

interpreted: JavaScript must be interpreted, at least partly, to comply with its standard. This allows for an `eval()` function to evaluate strings as JavaScript expressions.

JavaScript is a dynamic and very flexible language. The aforementioned features contribute to that fact, along with some less important features which are mentioned in the following paragraphs, along with ways to work with them or work around them.

For building a framework like the one at hand[2], some of JavaScript's features prove disadvantageous and have to be mitigated:

Variables in JavaScript are global, unless they are preceded with the keyword `var`. In that case a variable is local. However, there is is no block scope, meaning that a variable is visible in the entire (innermost) function surrounding it. Practically e.g. the variables from within a for loop (i.e. block) are visible in the surrounding function and defined until the end of the function body.

[2]The implementation consists of about 4950 lines of code of JavaScript. Counting was done on physical lines of code, which includes comments.

This means care must be taken with regard to not overlook any declarations with a missing `var`. Moreover these declarations must not clash with declarations in nested blocks, because these have no scope of their own, as was pointed out.

JavaScript has no notion of packages or namespaces to avoid name clashes on the global level.

To remedy this, a JavaScript object may be created in the global scope serving as a package. All parts of the framework are subsequently assigned to properties of that object, effectively becoming sub-packages.

JavaScript functions are variadic. Variadic or *variable arity* means that functions accept an arbitrary number of parameters, even if a fixed number of formal parameters have been defined. Practically e.g. , this means that more parameters might be passed to a function than its number of formal parameters, or worse, less than the formal parameters are passed to the functions. In the latter case the remaining formal parameters are undefined. In the former case the extra parameters can be accessed through a special `arguments[]` array.

To remedy the second case, where some parameters are undefined in a function, assertions are used in this work. If an assertion is violated, either an exception is thrown or execution is aborted altogether. Exceptions are part of JavaScript since the above-mentioned standard version ECMA-262, along with a try-catch syntax. Throwing exceptions is mostly used for input data like rules from the user, which might be erroneous or missing. Halting execution of the programme is only done in extreme cases. Those cases are subsequently tried to be ruled out in the implementation phase by testing all occurrences of calls to the function. Tests include checking for the number of parameters and whether they are defined and whether they contain the correct types. By propagating the constraints backwards through the function calls, correctness in this regard can be proven manually at implementation time.

JavaScript is interpreted. In many implementations of JavaScript engines this means that its expressions are evaluated step by step. Optimizations like *common subexpression elimination* (cf. [GoWa84, Section 13.2.1]) from multi-pass compilers are not performed.

To remedy this, towards a performance gain, some optimizations in JavaScript may be done manually by the programmer. Two examples are caching of access to deeply nested attributes and factoring out constant values from a loop. The former is necessary because accessing nested attributes is expensive in JavaScript. Caching is achieved by the programmer by assigning the value of such an attribute to a temporary variable once. The temporary variable is subsequently used for further access to its value. In further expressions using this value, the technique also provides better readability if the new identifier has been chosen well. The latter of the two examples, factoring out constant values from a loop, is particularly helpful in the often used loop pattern of

going through an array up to a counter of `array.length`. If the upper bound for the counter is part of the loop condition, it is evaluated in each cycle. When the length of the array is not changed in the loop this can easily be assigned to a temporary variable before the loop, or in the initialization expression of the `for`-statement, i.e. before the first semicolon.

JavaScript is single-threaded. A Web application can therefore only use one thread.

To remedy this, asynchronous programming is used. This is a programming style which is encouraged by JavaScript. Application logic is divided into event handlers which are queued in the global event queue of JavaScript. The queue lists all JavaScript event handlers which are active and want to run. Running code must yield to the next queued item before that may run. Such an execution model is cooperative, there is no preemption of running code.

This ends the list of problems and associated solutions resulting from programming in JavaScript in general. The following sections describe further methods and approaches which are used and highlight some noteworthy aspects of the implementation which are special to JavaScript or otherwise worth reporting.

5.2 JavaScript Object Notation

JavaScript Object Notation (JSON) is a lightweight format to represent data. It is used to form the rule language proposed in this work. Syntactically it is a subset of JavaScript. The subset contains all literal values, starting with `true`, `false` and `null`. Also included are string literals, number literals and most importantly the object and array literals, denoted in curly braces or square brackets, respectively. The array literal facilitates ordered lists, tuples, and the like. The object literal facilitates records, hash maps, unordered sets, etc. By nesting these primitives, many complex data types are available.

What is missing from the JavaScript language, is any kind of statement, functions, and expressions containing operators. The primitives of JSON have to be from among the previously mentioned literals or compositions thereof. JSON is therefore purely a data interchange format, there are no executable parts.

In the proposed rule language it is in some places necessary to deliver executable code. This means some parts of the rules must contain JavaScript statements. For this purpose strings are used to contain the statements. As it is mentioned above, JavaScript contains an `eval()` function. It evaluates strings at run-time, treating them like expressions or statements. In this way arbitrary actions can be invoked, specified by the rules.

JSON used for the rule language in this thesis was published as a Request for Comments RFC 4627 [Croc06]. Some differences and advantages over XML are discussed in Section 4.4. The main advantages are light weight and ease of integration with JavaScript applications, like Rich Internet Applications.

JSON is JavaScript and therefore can be evaluated to JavaScript variables directly by the `eval()` function. However, because JSON is a data interchange format, it must not contain any executable statements. Hence, a dedicated parsing function is used to enforce these restrictions on the input. Violating expressions which are not valid JSON are ignored or raise an error, depending on the implementation.

The fact that a JSON data type may not contain any executable code does not prevent the rule language from containing JavaScript code as one or more strings. Such strings safely pass the validation phase and can be investigated and evaluated later. However, they may not perform any potentially harmful operations by being evaluated accidentally during the JSON evaluation.

There are several pure JavaScript implementations of parsers on a Web site[3] maintained by Douglas Crockford, the inventor of JSON and author of the mentioned RFC. For this work the reference implementation from http://www.json.org/json2.js is used. It relies on `eval()` but it breaks down the expressions as far as necesary before evaluating the remaining sub-expressions so that no unwanted parts should be evaluated.

There is another implementation for even higher security requirements which is completely setting aside `eval()`. It simulates a complete parser using only string manipulation and regular expressions. However, it is much slower and the added security was not needed for this work, because no untrusted input was handled.

Apart from the current pure JavaScript parsers for JSON, there will be native parsers included in future Web browsers which are even faster. At the time of this writing, however, none of them are available yet, so the above mentioned reference implementation in JavaScript is chosen.

5.3 Comet Architecture

State of the art event-driven RIAs rely either on the Comet architecture introduced by Alex Russell in 2006 [Russ06] or on polling strategies like fetching RSS feeds to receive event data from remote Internet sources.

Because employing a push approach is more resource-efficient and facilitates timelier reactions to events, a push approach is used in the implementation of server-triggered events for this work.

RSS feeds also provide events like news items, etc, with a pull approach. The number of RSS feeds on the Web is huge and certainly cannot be ignored as a rich, pre-existing source of information. However, as mentioned above, for this work the feasibility of a push approach is shown for its technical advantages.

The Comet architecture is an umbrella term for different techniques of pushing data from a Web server to a Web client. One technique uses a hidden `iframe` in the page

[3] Introducing JSON. Online Resource. http://www.json.org/

displayed at the client. In the iframe another Web page is requested but the server only responds with data incrementally when events are to be transferred. This is the basic Comet technique for keeping a long-lived connection open between client and server. There are other techniques which are more advanced but do not work in every Web client implementation. The techniques can be combined, however, and employed according to the capabilities of the encountered client.

For this work, a Comet product called *Lightstreamer*[4] is used. It combines several approaches for opening a long-lived connection from the client side. On the server side it implements a Java-based Web server.

Lightstreamer originated in software for sending events of financial data. This is fitting but not limited to the financial market example which is implemented for this thesis. The actual data transferred from server to client for this example are generated stock market events. Per default these event occur at random intervals, but is also possible to create them in a fixed frequency and thus bandwidth.

Events from Lightstreamer are handled on the client side by an event handler function. This function serves as adapter which feeds simple event objects into the detection graph. From this point on the server-triggered events may participate in complex events, trigger rules or be discarded altogether if they do not match an event pattern. This described the transportation and delivery of events from remote Internet servers.

As a side note on server-pushed events, early drafts on the standard HTML 5 include a proposal for "Server-sent DOM events" in [HiHy08, Section 6.2]. For the next generation of Web standards this might yield a native way of subscribing to server-pushed events, independent of programming languages and implementations.

5.4 Graph-based Complex Event Detection

Other simple event types, apart from the explained server-triggered events are clock-triggered temporal events and user-triggered Document Object Model (DOM) events.

Temporal events are created using JavaScript's `window.setTimeout()` or `window.setInterval()` functions. Both are passed a callback function which is invoked when the time-out or interval is fired. The duration is passed as a second parameter in milliseconds. The difference between the functions is that the latter invokes its duration time periodically, invoking the callback function repeatedly until the interval is cleared.

Clearing of the time-out or interval is necessary when a temporal event is removed from the graph. Then its system timers must be cleaned up. Otherwise the Web

[4] Lightstreamer product Web site. Online Resource. http://www.lightstreamer.com/

browser will retain a reference to the callback function and invoke it in some time in the future when the function has no proper context anymore.

Every invocation of setInterval() or setTimeout() returns an integer identifier which must be used to deregister the timer later. A periodic event P, for example, might have the following Snoop expression: `P(E1, [2 secs], E2)`. This event triggers a temporal event every two seconds from the time an event E1 is detected. The periodic events will stop when an event E2 is detected.

Upon detection of the first event E1, the node for the P-event invokes setInterval() with a callback function containing a reference to the P-event node and a time of two seconds. The callback will later invoke a trigger() function back on the P-event node to notify it of an occurrence of the recurring two-seconds-event.

When another event E1 is detected, a parallel, overlapping series of P-events starts. Therefore, setInterval() is called again, with the predefined two seconds. The reference back to the P-event node also is the same. However, a reference to the event E1 is added to distinguish the callbacks of the different series of events. The integer identifiers from the setInterval() function are stored in an array. When the first event E2 is detected, the oldest E1 event is removed, and its timer is cancelled using the first integer identifier from the mentioned array.

So much for complications of detecting overlapping instances of events while at the same time carefully registering and deregistering timers with the JavaScript run-time system.

DOM events are another class of simple events. They are detected on nodes of the internal tree in the browser representing the elements of a Web page, for example of a RIA. The specified DOM node defines *where* an event is detected. The included event type defines *what* incident is detected.

Typical types of events from DOM elements are: *blur, change, click, dblclick, focus, keydown, keypress, keyup, load, mousedown, mouseout, mouseover, mouseup, select, submit and unload.*

The jQuery[5] library adds some more events to this list. The library is used in this work because it allows the subscription to the mentioned event types on arbitrary DOM nodes in a declarative fashion.

No functions need to be called to register listeners. The desired node is provided by a so-called CSS selector from the Cascading Style Sheets language or an XPath expression. Sets of nodes are also supported as the result of selectors or XPath expressions. The specified event is then subscribed on all of the matching nodes. Unsubscribing works using the selector or XPath in the same way.

This concludes noteworthy features of event types. The event graph itself posed an interesting problem when new rules were added to the engine, explained in the following.

[5] jQuery JavaScript library. Online Resource. http://jquery.com/

A new rule, if it is a reaction rule, contains an event pattern which must be added to the event graph. The new event pattern is first transformed into a tree of appropriate nodes which later fulfil the actual event detection. The tree is then integrated with the existing graph but nodes must be reused if they serve the same pattern or sub-pattern to save memory and multiple detection of identical events.

Event nodes have different types corresponding to their purpose, i.e. operators the represent. Also, they might be specified by the user to run in different detection contexts, resulting in different semantics. Thirdly their children nodes must be identical because the output of a certain node is always a function of its input, which is propagated upwards from these children.

Because incoming events are fed into the graph at the bottom, the leaves there are the smallest building blocks to start determining equality of larger graphs. When equality is established, a node may be shared, and the new node transfers all its attached parent nodes to the existing equal node to supply them with input events. The new node is then not used any further.

Simple event nodes at the leaves of the graph are atomic, meaning they do not have children. Therefore, the identity of a new node and an exiting node can be determined without recursion. In the case of temporal events it is the `timeString` which must be identical, for DOM events is the DOM selector and DOM event type, etc.

After adding or sharing the lowest level of nodes, the next level is processed and so on, in a manner of merging the new tree into the existing graph, bottom-up. When an inner node is processed which has shared children, the inner node must also be tested upon whether it can reuse an existing node. On the other hand, if at least one of its children is new, the node need not be tested because having one unique, novel child makes the node singular in itself.

When an inner node is tested for equality with existing nodes, it is compared with its siblings. Equality is now defined as the node type, the detection context and possibly some type specific values like the parameters of P or P* nodes. So far equality of the node function is tested, but as it is mentioned above, the input must also be equal before a new node can really be discarded and be fully replaced by an existing node.

Equality of children, however, needs not to be tested recursively downwards from this point. Because the tree has been merged bottom-up, all nodes below are already either unequal, or otherwise *identical* after being merged into one reused node. This means that for equality of children, only referential identity must be verified, which is very efficient.

This means that merging a new event tree into the existing graph entails only the complexity of traversing the graph layer after layer but without having to undergo recursive computations from each layer down to the leaves.

Reduction of the event graph is algorithmically speaking easier than adding to the graph. An event expression which shall be deleted from the graph has a top node.

That node is referenced from the rule repository with its corresponding reaction rule, for example.

On this node the method `removeBranch()` is subsequently invoked. The method is defined on all types of graph nodes and may therefore recursively call itself on all the children. When a node is otherwise not shared by other nodes, removeBranch() finalizes the event node and descends to the children. Finalizing might include clearing all system timers which were registered or for a simple event also unlinking itself from the list of all simple events, where the leaves are stored.

The method removeBranch() does nothing, when a node has further parents which might rely on its output or when further rules are attached to the node awaiting to be triggered. In these cases the node is either part of higher event expressions or in the case of the attached rule is used by another rule listening on the same event part. Then the recursion terminates before it reaches the leaves.

5.5 Rule Engine using Rete

The Rete network is constructed from the top downwards, contrary to the event graph. This is because working memory elements (WMEs) enter the Rete network at a single, top node. As with the event graph, equal nodes must be shared. Equality is likewise determined by the function of a node combined with its input, meaning its predecessor nodes.

An example Rete network is shown in Figure 4.5 on page 45. The figure shows the top node in the top left corner. The connected alpha network contains checks on single objects, i.e. WMEs and stores applicable objects in its alpha memories. The beta network subsequently performs joins after all single object checks are verified before in the alpha network.

Constructing the Rete network from the rule specification is done as follows. Each class pattern is first converted into a series of consecutive alpha nodes. There are different types of alpha nodes forming sub-classes of `Node.Alpha`, cf. Figure 4.6 on page 46.

These alpha nodes for example perform checks on the class of an object, the existence of an attribute of an object, or comparisons with the values of attributes, etc.

On adding it to the network, each alpha node is linked to its predecessor, checking whether an equal node is already among the successor nodes and sharing it, if so. After the single object checks are completely represented in the network, an alpha memory is added in the end to store the output.

To create the successive beta network, joins are gathered from the rule specification. Every free variable occurring in more than one object pattern is invoking a join. Joins are then ordered in pairwise joins by variable and by input memory.

Beta nodes are then created with the necessary join predicates and attached to the matching alpha memories. A join predicate or `Test` is a JavaScript function. It is selected from a hash map of predefined comparator functions which are selected by the comparator specification in each rule, cf. Subsection 4.4.2.

Comparator functions include wrappers for the built-in comparators from JavaScript like `<, >, <=, >=, ==, !=` and like `===, !==` which do not perform type coercions like their two-letter counterparts do. Also the JavaScript special operator `typeof` is available, which allows checks for the types of objects and primitives. Furthermore, there are set operators and the operator for the ontology. Adding more functions to the hash map here provides simple extensibility for the rule framework.

The comparator functions are two-parameter functions with boolean result because they are used as join predicates. The functions are are stored in the `Test` objects in join nodes.

A join node has a beta memory as one input and an alpha memory as another. The beta memory supplies tokens which are lists of objects satisfying preceding joins, cf. Subsection 4.2.2. The alpha memory supplies plain objects (in the form of WMEs) which must match the other objects in the token according to the join predicates.

After finishing all joins from the dummy beta memory to the end, a production node is added to the network. Such a node is a beta memory containing finished tokens representing a complete join. Each such token resembles a fully matched pattern and therefore a rule action is triggered from the production node.

This concludes the build-up of the Rete network, a dynamic graph representing the rules currently active at any given time of the application life-cycle.

6

Summary and Conclusions

Although the field of Complex Event Processing is not new, it is still undergoing research and many questions remain open. This thesis provides a solution for event processing on a final tier of enterprise applications, the Web-client. By taking part in the event processing locally, it can become an active member in a larger event-driven architecture, rather than just a viewing device or passive end-point.

To ease the necessary programming effort of detecting events, a declarative rule language was introduced which allows several types of reactive rules. The rule language with its accompanying rule engine also serves a purpose to adopt the Business Rules Approach to programming on the final tier, the client. In doing so, the experience of non-IT professionals can be used who are accustomed to authoring business rules for server applications. Also, rules might be shared with the remaining business infrastructure, along with a common vocabulary in the form of an ontology.

To gain the most out of a client-side rule framework, a large rule base should be established. Existing business rules could be translated into client-side rules where applicable and Web-based repositories should be created to store and retrieve exiting rules.

Other than that the rule framework is complete and should provide a start for declarative programming of adaptive, reactive Rich Internet Applications including Complex Event Processing.

Grammar of the JSON Rule Language

This is the grammar for the JSON Rule Language, a lightweight language for specifying event-condition-action rules. The language is designed with Rich Internet Applications in mind, but should not be limited to them. It is developed as part of this theses, cf. Section 4.4 on page 51.

The grammar is specified in (extended) Backus-Naur Form (BNF) with minor additions for the parser generator tool ANTLR[1].

The syntax of ANTLR's BNF grammars is fairly straightforward. Productions consist of a left-hand side (LHS) and right-hand side (RHS). They are specified as: `LHS : RHS;` Terminal symbols in the RHS are enclosed in single quotes. As mentioned above, the ANTLR tool adds some syntax elements. Namely this entails the declaration `grammar JsonRules;` at the top of the grammar file, parser directives like `$channel=HIDDEN;` and the keyword `fragment`.

The directive `$channel=HIDDEN;` in the production `WS` for white space, tells the parser to ignore tokens produced by this rule. Tokens for `WS` are therefore only used in the lexer, mainly in the lexer rules grouping identifiers with the colon which must follow them.

The mentioned keyword `fragment` introduces a part of a token which can be reused in different tokens to make productions more readable. The fragments are in-lined by in the respective tokens before a lexer is generated. Thus, no actual tokens are produced for fragments, and fragments therefore are purely a short hand.

[1] ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/

The grammar is in LL(k). All left recursion is eliminated from its productions. The tool ANTLR used for creating and testing the grammar is an LL(k) parser generator, which is the reason for choosing this particular grammar class.

```
1 //
2 // ANTLR Grammar for the JSON-Rules language
3 //
4 grammar JsonRules;
5
6 // Event-Condition-Action
7 ruleFile : '{' (ruleSetMetaData ',')? (eventBase ',')?
8     (conditionBase ',')? (actionBase ',')? RULES ruleSet '}';
9
10 ruleSet : '[' (reactionRule (',' reactionRule)*)? ']';
11
12 eventBase : EVENTBASE
13     '{'
14     ((QUOTEDSTRING ':' '{' eventExpression '}')
15     (',' QUOTEDSTRING ':''{' eventExpression '}')*)?
16     '}';
17
18 conditionBase : CONDITIONBASE '{' '}';
19
20 actionBase : ACTIONBASE '{' '}';
21
22 reactionRule : '{' (ruleMetaData ',')? (eventPart ',')?
23         (conditionPart ',')? actionPart '}';
24
25 eventPart : EVENT event;
26
27 conditionPart : CONDITION
28         '[' (conditionElement (',' conditionElement)*)? ']';
29
30 actionPart : ACTION '[' action (',' action)* ']';
31
32 ruleMetaData : META meta;
33
34 ruleSetMetaData : META meta;
35
36 // Event
37 event
38     :   '{' (eventExpression | eventName) '}';
39         // Either an elaborated complex event expression or
40         // the name of a predefined or simple event.
41         // Predifined events are specified in the eventBase.
```

```
42
43 eventExpression
44     :   eventTypeExpression
45         // Used with the ANY(m, E1,...En) event:
46         (',' M NumericLiteral)?
47         // Used for everything except TEMPORAL and DOM:
48         (',' eventChildrenExpression)?
49         // Used with P, P*, PLUS and TEMPORAL events:
50         (',' timeStringExpression)?
51         // Used with MASK event:
52         (',' maskExpression)?
53         // Used with DOM event:
54         (',' domExpression)?
55         // Used with P and P* events:
56         (',' paramsExpression)?
57         // Event detection context:
58         (',' contextExpression)?;
59
60 eventTypeExpression
61     :   TYPE QUOTEDSTRING;  // ('"OR"' | '"AND"' | '"ANY"' |
62                         // '"SEQ"' | '"A"' | '"A*"' |
63                         // '"P"' | '"P*"' | '"NOT"' |
64                         // '"PLUS"' | '"MASK"' |
65                         // '"TEMPORAL"');
66 maskExpression
67     :   MASK '{'
68             // Optional when maskType is "JAVASCRIPT".
69             (maskTypeExpression ',')?
70             argumentsDefinition
71         '}';
72
73 domExpression
74     :   SELECTOR QUOTEDSTRING ','
75         EVENT QUOTEDSTRING;
76
77 paramsExpression
78     :   PARAMETERS '{'
79             QUOTEDSTRING ':' QUOTEDSTRING
80                 (',' QUOTEDSTRING ':' QUOTEDSTRING)* '}';
81
82 contextExpression
83     :   CONTEXT QUOTEDSTRING;   // ('"RECENT"' | '"CHRONICLE"' |
84                         // '"CONTINUOUS"' |
85                         // '"CUMULATIVE"')
```

```
86 maskTypeExpression
87     :   MASKTYPE QUOTEDSTRING;
88
89 argumentsDefinition
90     :   ARGUMENTS '[' QUOTEDSTRING (',' QUOTEDSTRING)* ']';
91
92 timeStringExpression
93     :   TIMESTRING QUOTEDSTRING;
94
95 eventName
96     :   NAME QUOTEDSTRING;
97
98 eventChildrenExpression
99     :   CHILDREN '[' event (',' event)* ']';
100
101 // Condition
102 conditionElement
103     :   '{'
104         // either the class and name may be left out...:
105         ((
106             (CLASS QUOTEDSTRING ',')?
107             (NAME QUOTEDSTRING ',')?
108             (VARDEF QUOTEDSTRING ',')?
109             FIELDS '[' (conditionField
110                 (',' conditionField)*)? ']'
111         ) |
112         // ... or the fields may be left out:
113         (
114             (CLASS QUOTEDSTRING | NAME QUOTEDSTRING |
115                 CLASS QUOTEDSTRING ',' NAME QUOTEDSTRING)
116             (',' VARDEF QUOTEDSTRING)?
117         ))
118         '}';
119
120 conditionField
121     :   '{'
122         FIELD QUOTEDSTRING
123         (',' COMPARATOR QUOTEDSTRING (
124             (',' LITERAL flatLiteral) |
125             (',' VARIABLE QUOTEDSTRING) |
126             (',' FIELD2 QUOTEDSTRING))
127         )?
128         (',' VARDEF QUOTEDSTRING)?
129         '}';
```

```
130
131 // Action
132 action  :   '{'
133         TYPE QUOTEDSTRING  // '"JAVASCRIPT"' | '"EVENT"'  | '"ASSERT"' ...
134         (',' FUNCTION QUOTEDSTRING)? // Used with JAVASCRIPT
135         (',' TRIGGER QUOTEDSTRING)? // ... with EVENT
136         (',' CLASS QUOTEDSTRING)? // ... with ASSERT, optionally
137         (',' NAME QUOTEDSTRING)? // ... with ASSERT, RETRACT and MODIFY
138         (',' OBJECT literal)? // ... with ASSERT
139         (',' MODIFY QUOTEDSTRING)? // ... with MODIFY
140         '}'
141         ;
142
143 // Metadata
144 meta
145     :   '{'
146         (RULE | RULESET) QUOTEDSTRING
147         (',' AUTHOR QUOTEDSTRING)?
148         (',' VERSION (QUOTEDSTRING | NumericLiteral))?
149         (',' keyValueAssignment)*
150         '}';
151
152 keyValueAssignment
153     :   QUOTEDSTRING ':' flatLiteral;
154
155 flatLiteral
156     :   QUOTEDSTRING | NumericLiteral | 'true' | 'false' | 'null';
157
158 literal :   object | array | flatLiteral;
159
160
161 object  :   '{' (QUOTEDSTRING ':' literal
162         (',' QUOTEDSTRING ':' literal)*)? '}';
163
164 array   :   '[' (literal (',' literal)*)? ']';
165
166
167
168 // Lexer rules, i.e. tokens:
169 META    :   '"meta"' WS* ':';
170 ACTIONBASE: '"actionBase"' WS* ':';
171 EVENTBASE:  '"eventBase"' WS* ':';
172 CONDITIONBASE:  '"conditionBase"' WS* ':';
173 RULES   :   '"rules"' WS* ':';
```

```
174 EVENT    :   '"event"' WS* ':';
175 CONDITION:   '"condition"' WS* ':';
176 ACTION   :   '"action"' WS* ':';
177 NAME     :   '"name"' WS* ':';
178 TYPE     :   '"type"' WS* ':';
179 M    :   '"m"' WS* ':';
180 MASK     :   '"mask"' WS* ':';
181 MASKTYPE:    '"maskType"' WS* ':';
182 ARGUMENTS:   '"arguments"' WS* ':';
183 SELECTOR:    '"selector"' WS* ':';
184 PARAMETERS:  '"parameters"' WS* ':';
185 CONTEXT :    '"context"' WS* ':';
186 TIMESTRING:  '"timeString"' WS* ':';
187 CHILDREN:    '"children"' WS* ':';
188 CLASS    :   '"class"' WS* ':';
189 FIELDS   :   '"fields"' WS* ':';
190 FIELD    :   '"field"' WS* ':';
191 COMPARATOR:  '"comparator"' WS* ':';
192 VARDEF   :   '"vardef"' WS* ':';
193 LITERAL :    '"literal"' WS* ':';
194 VARIABLE:    '"variable"' WS* ':';
195 FIELD2   :   '"field2"' WS* ':';
196 FUNCTION
197      :   '"function"' WS* ':';
198 TRIGGER :    '"trigger"' WS* ':';
199 OBJECT   :   '"object"' WS* ':';
200 MODIFY   :   '"modify"' WS* ':';
201 RULE     :   '"rule"' WS* ':';
202 RULESET :    '"ruleSet"' WS* ':';
203 AUTHOR   :   '"author"' WS* ':';
204 VERSION :    '"version"' WS* ':';
205 QUOTEDSTRING
206      : '"' ( EscapeSequence | ~('\u0000'..'\u001f'
207      | '\\' | '\"' ) )* '"';
208 fragment EscapeSequence
209          : '\\' ( UnicodeEscape |'b'|'t'|'n'|'f'|'r'
210          |'\"'|'\''|'\\');
211 fragment UnicodeEscape
212      : 'u' HexDigit HexDigit HexDigit HexDigit;
213 NumericLiteral
214      : DecimalLiteral | HexIntegerLiteral    ;
215 fragment HexIntegerLiteral
216      : '0' ('x' | 'X') HexDigit+ ;
217 fragment HexDigit
```

```
218     : DecimalDigit | ('a'..'f') | ('A'..'F');
219 fragment DecimalLiteral
220     : DecimalDigit+ '.' DecimalDigit* ExponentPart?
221     | '.'? DecimalDigit+ ExponentPart? ;
222 fragment DecimalDigit
223     : ('0'..'9');
224 fragment ExponentPart
225     : ('e' | 'E') ('+' | '-') ? DecimalDigit+ ;
226 WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;};
227
```

Bibliography

[ABCG05] Silvestre Losada Alonso, Jose Luis Bas, Sergio Bellido Jesús Contreras, and Jose Manuel Gomez. D10.7: Financial Ontology. Deliverable, Data, Information, and Process Integration with Semantic Web Services: Project DIP, October 2005.

[AdCh06] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.

[Alla02] Jeremy Allaire. Macromedia Flash MX—A next-generation rich client. Technical report, Macromedia, March 2002.

[Bass07] Tim Bass. Mythbusters: Event Stream Processing versus Complex Event Processing. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 1–1, New York, NY, USA, 2007. ACM.

[Bato94] Don Batory. The LEAPS Algorithms. Technical report, University of Texas at Austin, Austin, TX, USA, 1994.

[BaVl97] N. Bassiliades and I. Vlahavas. DEVICE: Compiling production rules into event-driven rules using complex events. *Information and Software Technology*, 39(5):331–342, 1997.

[Behr94] H. Behrends. An Operational Semantics for the Activity Description Language ADL. Technical report, Universität Oldenburg, June 1994. Technical Report TR-IS-AIS-94-04.

[Bers02] Bruno Berstel. Extending the RETE algorithm for event management. In *Proc. Ninth International Symposium on Temporal Representation and Reasoning TIME 2002*, pages 49–51, Washington, DC, USA, 7–9 July 2002. IEEE Computer Society.

[BFKM85] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[BiJo87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[Bole01] Harold Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *INAP*, pages 124–139, 2001.

[BrEc06] F. Bry and M. Eckert. Twelve theses on reactive rules for the web. *Proceedings of the Workshop on Reactivity on the Web, Munich, Germany*, 2006.

[CCBF07] Giovanni Toffetti Carughi, Sara Comai, Alessandro Bozzon, and Piero Fraternali. Modeling Distributed Events in Data-Intensive Rich Internet Applications. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *WISE*, volume 4831 of *Lecture Notes in Computer Science*, pages 593–602. Springer, 2007.

[Chak97] Sharma Chakravarthy. SENTINEL: An Object-Oriented DBMS With Event-Based Rules. In Joan Peckham, editor, *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 572–575. ACM Press, 1997.

[CKAK94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 606–617, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[Croc06] Douglas Crockford. RFC4627: JavaScript Object Notation. Technical report, IETF, 2006.

[CuRD93] C. Culbert, G. Riley, and B. Donnell. CLIPS Reference Manual Volume 1, Basic Programming Guide, CLIPS Version 6.0. *Software Technology Branch, Lyndon B. Johnson Space Center, NASA*, 1993.

[DaBM88] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 129–143, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[Door95] Robert B. Doorenbos. *Production matching for large learning systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[Etzi08] Opher Etzion. On Event Processing Research Challenges. Online Article. http://epthinking.blogspot.com/2008/05/on-event-processing-research-challenges.html, May 2008. Last visited: July 2008.

[Forg82] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GaAu02] Antony Galton and Juan Carlos Augusto. Two Approaches to Event Definition. In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 547–556, London, UK, 2002. Springer-Verlag.

[GaDi94] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using Petrinets. In *Proc. Fourth International Workshop on Active Database Systems Research Issues in Data Engineering*, pages 2–9, 1994.

[GeJS92] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[GeJS93] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. COMPOSE: A System For Composite Specification And Detection. In *Advanced Database Systems*, pages 3–15, London, UK, 1993. Springer-Verlag.

[GoWa84] Gerhard Goos and William Waite. *Compiler Construction.* Springer, Jan 1984.

[Grub93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[HiHy08] Ian Hickson and David Hyatt. HTML 5—W3C Working Draft. Online Resource. http://www.w3.org/TR/html5/, June 2008. Last visited: July 2008.

[HiVo02] A. Hinze and A. Voisard. A parameterized algebra for event notification services. *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings.Ninth International Symposium on*, pages 61–63, 2002.

[Jens92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use.* Springer, 1992.

[Luck01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Luck06] David C. Luckham. What's the Difference Between ESP and CEP? Online Article. http://complexevents.com/?p=103, August 2006. Last visited: July 2008.

[LuSc07] David C. Luckham and Roy Schulte. Event Processing Glossary. Online Resource. http://complexevents.com/?p=195, May 2007. Updated: May 2008. Last visited: July 2008.

[Macl99] Saunders Maclane. *Algebra*. American Mathematical Society, Providence, 1999.

[Mart99] J.P. Martin-Flatin. Push vs. pull in Web-based network management. *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, pages 3–18, 1999.

[McDa89] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1989. ACM.

[Mira90] D.P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1990.

[PaKB07] A. Paschke, A. Kozlenkov, and H. Boley. A Homogenous Reaction Rules Language for Complex Event Processing. In *International Workshop on Event Drive Architecture for Complex Event Process*, 2007.

[PNFJ+08] Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith Jr., Edson Tirelli, Fernando Meyer, and Kris Verlaenen. JBoss Rules User Guide 4.0.7. Online Documentation. http://www.jboss.org/drools/documentation.html, 2008. Last visited: July 2008.

[Ramc74] C. Ramchandani. ANALYSIS OF ASYNCHRONOUS CONCURRENT SYSTEMS BY TIMED PETRI NETS. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[Russ06] Alex Russell. Comet: Low Latency Data for the Browser. Online Article. http://alex.dojotoolkit.org/?p=545, March 2006. Last visited: July 2008.

[ScAS08] Kay-Uwe Schmidt, Darko Anicic, and Roland Stühmer. Event-driven Reactivity: A Survey and Requirements Analysis. In *SBPM2008: 3rd international Workshop on Semantic Business Process Management in conjunction with the 5th European Semantic Web Conference (ESWC'08)*. CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073), June 2008.

[SSST07] Kay-Uwe Schmidt, Ljiljana Stojanovic, Nenad Stojanovic, and Susan Thomas. On Enriching Ajax with Semantics: The Web Personalization Use Case. In *Proceedings of the European Semantic Web Confer-*

ence, ESWC2007, volume 4519 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2007.

[Tidw06] Jenifer Tidwell. *Designing interfaces*. O'Reilly, 1. ed. edition, 2006.

[ZiUn99] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, pages 392–399, Washington, DC, USA, Mar 1999. IEEE Computer Society.

Index

active database, 9
Activity Description Language, 16
ADL, 16
algebra, 21
algorithmic trading, 22, 49, 61
alpha network, 45
arc expression, 33
architecture
  event-driven, 2

backward-chaining, 16
beta network, 45

CCL, *see* Continuous Computation Language
Chronicle context, 11, 37, 39
classification
  faceted, 44
CLIPS, 25, 27, 62
closure, 68
Comet, 65, 71
Compose, 10
conceptualization, 19
content guard, 23
content-based matching, 23
context
  Chronicle, 11, 37, 39
  Continuous, 12
  Cumulative, 12
  Recent, 11
Continuous Computation Language, 12
Continuous context, 12
coupling mode, 28
crosscutting concern, 67
Cumulative context, 12

database
  active, 9
database trigger, 9
detection context, 11, 15, 24, 37
detection-based semantics, 11
detector, 40
DEVICE, 57
DOM event, 23
Drools, 58
dummy token, 47

ECMAScript, 67
eval(), 68
event, 23
  complex, 8, 23
  DOM, 23
  explicit, 27
  overlapping, 24, 33
  simple, 7, 23, 34
  temporal, 23
event algebra, 21
event base, 53, 55
event cloud, 8
event consumption, 11, 24, 37
event detection
  automaton-based, 13, 32
  graph-based, 14, 32
  Petri net-based, 13, 33
event history, 38
event mask, 10, 16, 23, 36
event order, 9
event pattern, 22
event selection, 11, 24, 33, 37
event source, 23
event specification, 9
event stream, 8

event type, 22
event-driven architecture, 2
explicit event, 27

faceted classification, 44
feed-back, 27
first-class functions, 68
forward-chaining, 17, 18, 25, 43
function
  variadic, 69

HiPAC, 9, 28

inheritance
  prototype-based, 68
initiator, 11, 40
interval-based semantics, 11, 22, 28, 34

JavaScript, 67

matching
  content-based, 23
message-passing, 7

New York Stock Exchange, 50
NYSE, *see* New York Stock Exchange

Ode, 10, 23
ontology, 19, 34, 49, 61
operator
  filter, 23
  guard, 23, 36
  logical, 22, 36
  mask, 23, 36
  temporal, 22, 36
OPS5, 25, 27, 62
overlapping event, 24, 33

performance, 24
Petri net
  coloured, 33
  timed, 33
production rule, 15, 18, 25
production system, 25, 27, 62
Prolog, 16
prototype-based inheritance, 68

reaction rule languages, 15
Reaction RuleML, 12, 16, 51, 58
Recent context, 11
rematch-based removal, 48
removal
  rematch-based, 48
  scan-based, 48
  tree-based, 48
Rete algorithm, 18, 43
RSS feed, 65

SAMOS, 13, 33
scan-based removal, 48
semantics
  detection-based, 11
  interval-based, 11, 22, 28, 34
Sentinel, 14
server-triggered event, 23
Snoop, 10, 14, 16, 22, 24, 32
SnoopIB, 11, 34
StreamSQL, 12
synchrony, 7

temporal event, 23
terminator, 11, 40
timeString, 42
token, 47
  array-form, 48
  dummy, 47
  list-form, 48
transition, automaton, 33
transition, Petri net, 33
tree, graph, 32
tree-based removal, 48
trigger, 25

user event, 23
user interface patterns, 53

variadic function, 69
virtual synchrony, 7

working memory, 18, 25, 44, 62, 63
working memory element, 25, 44, 49, 62