Nuno Alexandre Neves Cruz

Licenciado em Eng. Informática

Reactive Hybrid Knowledge Bases

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : Carlos Viegas Damásio, Prof. Associado,
Universidade Nova de Lisboa
Matthias Knorr, Pós-Doc,
Universidade Nova de Lisboa

Júri:

Presidente:	Nuno Manuel Robalo Correia
Arguente:	Salvador Pinto Abreu
Vogal:	Carlos Augusto Isaac Piló Viegas Damásio

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2014

Reactive Hybrid Knowledge Bases

Aos meus pais

Acknowledgements

It's a great pleasure to express my sincere gratitude to Dr. Carlos Damásio for giving me the opportunity of doing this work. Also, I would like to thank Dr. Matthias Knorr who was willing to serve as a second reviewer of this work. Dr. Damásio and Dr. Knorr shown the utmost predisposition for any doubt along the way, always providing an extremely enthusiastic and motivating work environment.
I thank my friends for all the endless support and inspiration through this journey.
My parents and sister receive my sincerest gratitude and love for giving me continuous support and encouragement through these months.

Abstract

Hybrid knowledge bases are knowledge bases that combine ontologies with non-monotonic rules, allowing to join the best of both open world ontologies and close world rules. Ontologies shape a good mechanism to share knowledge on the Web that can be understood by both humans and machines, on the other hand rules can be used, e.g., to encode legal laws or to do a mapping between sources of information.

Taking into account the dynamics present today on the Web, it is important for these hybrid knowledge bases to capture all these dynamics and thus adapt themselves. To achieve that, it is necessary to create mechanisms capable of monitoring the information flow present on the Web. Up to today, there are no such mechanisms that allow for monitoring events and performing modifications of hybrid knowledge bases autonomously.

The goal of this thesis is then to create a system that combine these hybrid knowledge bases with reactive rules, aiming to monitor events and perform actions over a knowledge base. To achieve this goal, a reactive system for the Semantic Web is be developed in a logic-programming based approach accompanied with a language for heterogeneous rule base evolution having as its basis RIF Production Rule Dialect, which is a standard for exchanging rules over the Web.

Keywords: Rules, Ontologies, Action rules, Semantic Web

Resumo

As bases de conhecimento híbridas são bases de conhecimento capazes de combinar ontologias com regras não monotonicas, possibilitando juntar o melhor das regras baseadas no pressuposto do mundo aberto e fechado. As ontologias são uma boa forma de partilhar conhecimento na Web e permitem que sejam entendidas tanto por humanos como por máquinas, enquanto que as regras permitem, por exemplo, codificar leis jurídicas ou fazer mapeamentos entre fontes de informação.

Tendo em conta a dinâmica que existe actualmente na Web, é importante que estas bases de conhecimento híbridas possam captar essa dinâmica e adaptarem-se. Para que isso seja possível, é necessário criarem-se mecanismos que suportem estas bases de conhecimento e ao mesmo tempo sejam capazes de monitorizar os fluxos de informação presentes na Web. Até à data, não existem tais mecanismos que permitam a monitorização de eventos e actualização automática de bases de conhecimento híbridas.

O objectivo desta dissertação é então criar um sistema que combine estas bases de conhecimento híbridas com regras reactivas, com o intuito de monitorizar eventos e executar acções sobre a base de conhecimento. Para atingir este fim, é implementado um sistema reactivo para a Web Semântica, usando uma abordagem baseada em programação em lógica acompanhado de uma linguagem para fazer evoluir bases de conhecimento híbridas baseada no RIF Production Rules Dialect, que é um standard para troca de regras sobre a Web.

Palavras-chave: Regras, Ontologias, Regras de acção, Web Semântica

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Throughout the last two decades, the Web has been evolving in a way to adapt to the various technological advances. In the beginning, we had a web with static or nearly static content, nowadays, on the other hand, its content is increasingly dynamic. This evolution is, in a way, related to the increase of connected devices and applications, e.g. sensors or smartphones that use the Web as a support to exchange of information. On YouTube, for example, in every minute about thirty hours of video are uploaded, similarly on twitter in the same period of time about 100,000 tweets[1] are sent. These two examples clearly demonstrate that today the Web is an enormous continuous flow of information where data, information and knowledge are perpetually under modification.

Given the limitations of the human being, it is impossible for us to process all the amount of activity present on the Web. Therefore, there is a need to use machines to monitor in real time what is happening, in order to obtain always up to date information, and to act.

Reactive systems are systems capable of acting accordingly to the detection of certain events [Har87]. These types of systems are adequate for dynamic environments where it is necessary to deal with a huge flow of information and act in real time. The *reactive systems* can be based on *reactive rules* and in particular *Event-Condition-Action (ECA)* rules which trigger actions as a response to the detection of events. These rules have the following general structure:

On event **If** condition **Then** action

[1] http://www.intel.com/content/www/us/en/communications/internet-minute-infographic.html

The idea of these rules is quite simple. When an *event* occurs, if some *condition* over the state holds, an *action* is executed. This kind of rules is useful to act on dynamic environments. The system receives the events as input from an external source and reacts by performing actions over the background knowledge, in order to evolve to another state.

Enrichment of applications with reactive rules is a topic that has been widely discussed in the database community. It is called active database systems ([GSS04]). These type of systems implement ECA rules over the database. Their objective is to monitor changes made to the database and act according to those changes. Nowadays, there are several commercial database systems which implement this functionality, such as Oracle Database[2].

The World Wide Web Consortium (W3C)[3] has already developed a way to exchange one type of *reactive rules*, the production rules, among rule engines, in particular Web rule engines. A production rule is a piece of knowledge organized along an *WHEN condition DO action* structure aimed to evolve the state of the system by executing an action, begin the action only applied from a state where the condition is true [BBBEP07]. To exchange kind of rules, W3C developed the RIF Production Rule Dialect [SMHP13] which is an XML standard for exchanging production rules. However, there is still missing a single format for exchanging event-condition-action rules among rule engines.

When Tim Berners-Lee first proposed the Web, he envisaged the exchange of information in form of web documents, in order to make that information always available in a single place and format. That request was well met and the result was a Web filled with information in the form of documents. However, the information on these documents is nothing more than simple individual web pages which, eventually, can be linked amongst themselves through hyperlinks. Therefore, the Web can be seen as an enormous graph of connected documents.

Nevertheless, if we, for example, search the web for "Venus", we obtain results such as the planet, the tennis player, the roman goddess, among others. The problem we face is that machines do not understand what we search for. They just return information which contains the keywords of our search. This makes us conclude that the Web is mostly a repository of documents for human consumption. Semantic Web[4] arises as a necessity to give meaning to the information available on the Web, so it can be understood both by human beings and machines [BLHL+01]. By extending the semantics also to the machines, simple tasks such as buying an online ticket, which can currently only be performed by human beings, would be automatically taken care of by a computer.

Whereas on the Web, data and its hyperlinks are all described with HTML, on the Semantic Web, data and its relations are expressed using W3C standards, e.g. RDF [KCM14]

[2] http://docs.oracle.com/cd/B28359_01/appdev.111/b31088/exprn_intro.htm
[3] http://www.w3.org/
[4] http://www.w3.org/standards/semanticweb/

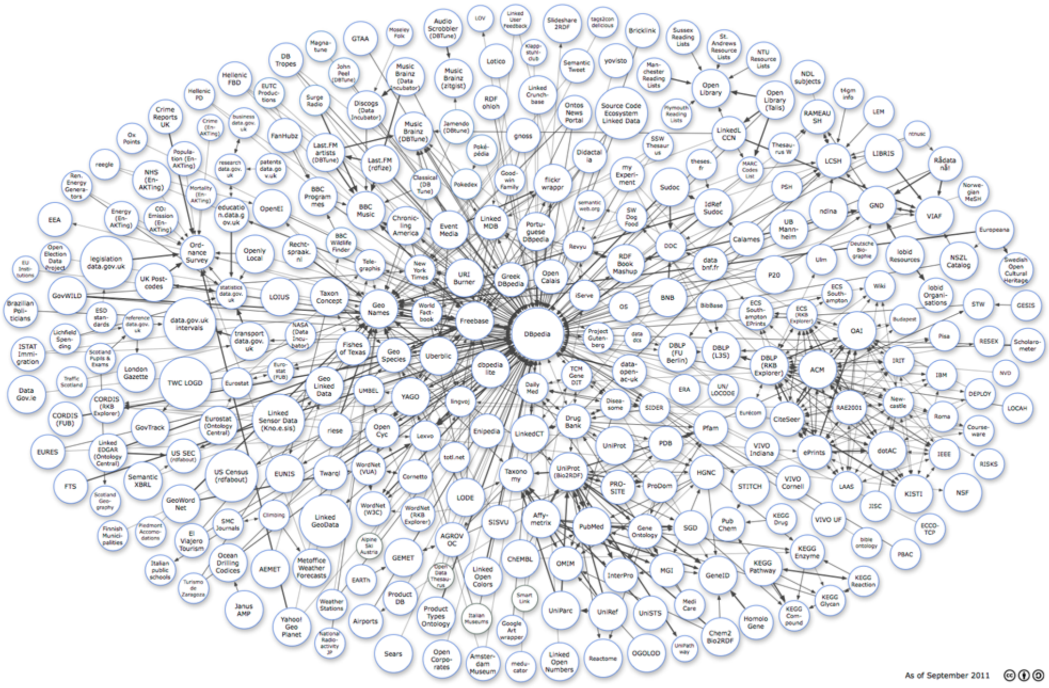


Figure 1.1: Overview of some datasources linked to DBpedia

or OWL [HKPPSR12]. This way, a machine can give meaning to the information on the Web and reason with this information. This new knowledge in the form of taxonomies (e.g. RDFS [BG14]) and ontologies (e.g. OWL) shape a good knowledge base, in the sense that certain domains of interest can be represented using concepts and relationships between concepts.

Semantic Web comes in layers, where the most basic element that is used to express knowledge is the Resource Description Framework (RDF). RDF is the W3C standard for encoding knowledge in the Semantic Web. It is used to represent metadata about Web resources and it is also used to represent information about things that can be identified on the Web. The next layer is RDF Schema (RDFS), which can be used to define the structure of the data. OWL is the highest level of expressity, where relations between classes can be formally modelled based on description logics [Bra07].

However, Semantic Web is not just about sharing data in the Web in a single format, but it is also about linking all the data. This collection of interrelated data is also referred to as Linked Data[5]. DBPedia[6] is an example of a large linked dataset. It contains all the information available on the Wikipedia and links that information with other datasets. Doing this, it allows for robots or Semantic Web engines to navigate through these datasets. Figure 1.1[7] depicts a diagram of all the datasets linked to DBpedia.

[5] http://www.w3.org/standards/semanticweb/data
[6] http://wiki.dbpedia.org/About
[7] http://wiki.dbpedia.org/Interlinking

The knowledge present on the Semantic Web can become even richer if combined with various rules, e.g., rules that encode laws. This combination allows to have a much richer knowledge, also called *Hybrid Knowledge Bases*.

The goal of Hybrid Knowledge Bases is to combine open world ontologies, such as the OWL-based ones, with closed world rule-based languages. Both ontologies and rules provide distinct strengths for the representation and interchange of knowledge in the Semantic Web and for applications of knowledge representation. According to [KAH11], the decision to rely on the Open World Assumption (OWA) appears to be a natural one considering the envisioned applications related to the World Wide Web: the absence of a piece of knowledge should not generally be taken as an indication that this piece of knowledge is false. However there are also cases where the Close World Assumption (CWA) is a more natural choice. The example described on [KAH11] shows the need for combining both close and open world assumption. For example, in a clinical domain, open world reasoning is needed to express that no assumptions about some result of a lab test can be made unless that result asserts a negative finding. However, the closed world assumption should also be used with data about medical treatment to infer that a patient is not on a medication unless otherwise stated. Combining rules and ontologies would yield a combination of the OWA and CWA, however this combination is not a trivial task because a simple combination is already undecidable. Nevertheless, there are several proposed approaches for combining rules and ontologies, such as the Well-founded Semantics for Hybrid Rules [DM07], Combining answer set programming with description logics [EILST08] and Reconciling Description Logics and Rules [MR10].

Combining these hybrid knowledge bases with reactive rules would allow us to have a new generation of knowledge-rich applications. As mentioned before, hybrid knowledge bases have a large application, e.g, in the domain of the life sciences, including medicine. If combined with reactive rules, these hybrid knowledge bases could be "aware" of the surrounding environment and self-adapt to the changes in that environment. For example, a hybrid knowledge base of some medical domain could be enriched with reactive rules that could, for example, be used in a monitoring device to help keeping updated the health status of some critical patient, according to the data received from the devices attached to the monitoring device. One can imagine these applications as being some agents that perceive the environment and evolve according to the state of its environment.

At present day, one of the challenges in the areas of Semantic Web and Knowledge Representation is to integrate *reactive rules* with hybrid knowledge bases. However, this integration has faced some problems. Firstly, as explained before, it has been difficult to integrate logic programming rules with ontologies due to the inherent differences of closed world assumption (CWA) and open world assumption (OWA). Secondly, *reactive rules* do not possess a declarative semantics that would facilitate the combination with hybrid knowledge bases.

The project ERRO – Efficient Reasoning with Rules and Ontologies – is a research project under development by the research centre CENTRIA at Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. This project aims to address the problems of 1) integration of reasoning rules with ontologies, in order to connect them on a single platform, and 2) develop a logic-based *reactive system* to monitor events and to act over hybrid knowledge bases, i.e., knowledge bases that combine ontologies and *reasoning* rules. Among the several approaches for combining ontologies and logic rules, the ERRO team chose the approach Hybrid MKNF knowledge bases [MR10], which builds on the logics of Minimal Knowledge and Negation as Failure (MKNF) [Lif91], with the semantic defined in [KAH11]. Its advantages lie in a smooth integration of ontologies and logic rules, and its comparably low complexity.

The main goal of this project is to develop a state-of-the-art platform and tools to provide efficient services for querying and updating hybrid knowledge bases and event monitoring to automate the execution of active rules. The platform will be fully integrated with Protegé[8]. Protegé is an open-source platform that provides users with tools to construct domain models and knowledge base applications with ontologies.

The work being developed within this thesis will focus on point 2) of ERRO - the development of a *reactive system*. The goal of this work is to develop the theory and the technology necessary to support event-based processing over hybrid knowledge bases on the Web, accompanied with implementations. Particularly, *Event-Condition-Action rules* will be used to implement such services supporting complex event detection, condition evaluation with respect to hybrid knowledge bases, and the execution of actions, e.g. insert/remove facts or axioms, over a hybrid knowledge base. This *reactive system* will be developed with a declarative semantics in order to foster the integration with hybrid knowledge bases in a single framework.

1.2 Contributions

The main contributions of this thesis are:

- A language for heterogeneous rule base evolution having RIF-PRD as its basis.
- An implementation of a *reactive system* using a logical programming paradigm, particularly using XSB-Prolog.
- Development of a Protégé plug-in to combine the *reactive system* with a hybrid knowledge base.
- Benchmarking of both the *reactive system* and the Protégé plug-in.

These contributions are indeed important because all combined, they contribute to the development of a new system capable of reacting over hybrid knowledge bases.

[8] http://protege.stanford.edu/

Since W3C does not provide a standard dialect to exchange ECA rules amongst Web rule engines, we propose a language to exchange this type of *reactive rules*. Also, the proposed language supports the combination of events, through the use of event algebras, which is useful to describe complex situation that are not described by an atomic event, such as a sequence of atomic events. This language is accompanied with a XSLT style sheet that transforms the XML syntax into a syntax compatible with the *reactive system*.

The *reactive system* then uses the transformed rules to trigger atomic and complex events and to perform actions over the hybrid knowledge base, which are triggered by the occurrence of those atomic or complex events. It is proved that this system is faster in detecting complex events than ETALIS implementation for XSB. Also, adding the reactive part to the CEP system does not have a significant negative impact on the system performance.

Finally, the Protégé plug-in is where a hybrid knowledge base is translated into a set of XSB rules, which are then sent to the *reactive system*. Also, the plug-in provides graphical interfaces to ease the user interaction. As far as we know, there is no such system capable of enhancing a hybrid knowledge base with ECA rules.

1.3 Document Structure

This document is structured into eight chapters: Introduction, State-of-the-art, Solution, Supported language, Reactive system, Protégé plug-in, Evaluation and Conclusion.

- **Introduction:** This chapter provides the motivation for this work, describes the objective of this thesis and also the expected contributions.
- **State-of-the-art:** Throughout this chapter is presented an overview of Event Algebra, Complex Event Processing, Stream Reasoning, Hybrid Knowledge Bases, ECA rules and Logic-based languages and tools.
- **Architecture** In this chapter, we briefly describe the whole architecture developed in this work, aiming to give the reader an intuition of what was developed.
- **Supported language** This is the chapter where we explain our supported language, based on RIF-PRD, for exchanging ECA rules.
- **Reactive system** This chapter is divided into two main sections, the *Complex event processing* and the *ECA rules processing*. In each section, we describe in detail the implementation of each part that composes the *reactive system*.
- **Protégé plug-in** In this chapter, we describe the implementation of the significant parts of the Protégé plug-in.
- **Evaluation** This is the chapter where we present the benchmarking results and discuss those results.

- **Conclusion** This is the final chapter where we present some conclusions about the work done and also some possible extensions to our work.

2

State of the Art

Since our goal is to develop a reactive system capable of monitoring events and performing actions over a hybrid knowledge base, it is necessary to make a brief overview of several related work. For the sake of the readability, this overview is structured into four main sections: **Events** (Section 2.1), **Hybrid Knowledge Bases** (Section 2.2), **Event Condition Action Rules** (Section 2.3) and **Logic-based Languages and Tools** (Section 2.4).

Taking into account that our system is concerned with the detection of events, we start this chapter by presenting several work about events. In order to better understand what an event is, atomic and complex events are characterized. After introducing the notion of an event, we present some approaches that aim to define an algebra to combine events. Next, we introduce Complex Event Processing systems whose goal is to detect complex events. Finally, we present some approaches to reason over an event stream. We will depart from the events and we will introduce hybrid knowledge bases, which represent the background knowledge of our system, in particular Hybrid MKNF. Since our system is built with reactive rules, particularly event-condition-actions rules, we will describe some existing approaches that implement those rules. The last section of this chapter lists logical-based rule technology. In particular, we will introduce RIF-PRD, ELK reasoner, XSB-Prolog, the NoHR plugin and the OWL API. All these technologies and tools will be used to build our hybrid reactive system.

2.1 Events

2.1.1 Atomic Events

To conceive the notion of atomic events, it is helpful to imagine a continuous time line. This line is divided into a number of segments where each segment represents the granularity level, i.e., in how many parts the time is divided. The time abstraction can be represented in several ways, e.g., it can be represented in hours, months or years. Thus from a system's point of view time is discrete rather than continuous. For the reactive system being developed within this thesis, it is also assumed a time discrete model.

An *atomic event* is something that happens in a specific point in time and has a zero duration. These events are always associated with a timestamp which represents the time of occurrence of the event. With respect to some system, atomic events can come from two sources: an external source, if they are generated outside the system, or an internal source, if they come from inside the system.

Besides the timestamp, events can be also associated with some information. This information can give more details about the occurrence of that event, e.g., data that identifies some entity, some numeric data or provenance data.

Definition 1. Let E be a finite set of identifiers that represent events. For each identifier $e_i \in E$, let $dom(e_i)$ be the domain of the data that characterizes the event and T be a finite set of integers that represent the time.

An atomic event can be represented as follows:

$$e(ts, d_1, \ldots, d_n)$$

where $e \in E$, $d_1, \ldots, d_n \in dom(e)$ and $ts \in T$. To better understand what an atomic event is, consider the following example:

Example 1. A temperature sensor is a device that measures the temperature from its surrounding environment. Usually these sensors send periodically their temperature measures to some remote system. Thus, an atomic event can be those measurements that are sent to the remote system and can be described as:

$$tmp\_sensor(ts, tmp)$$

where tmp_sensor is the event name, ts is the timestamp and the tmp is the numeric data that represents the temperature.

2.1.2 Complex Events

Complex events are events that can be composed of atomic events and/or other complex events. A complex event can be seen as a story, where a set of events are combined to

complete the whole story. They are useful to model events that do not occur instantaneously but occur in an incremental way over a period of time.

The main difference between an atomic event and a complex one is that the later can not occur in a single point in time, i.e., the time of occurrence is defined by the first and the last event contributing to its detection. However, as in the atomic events, complex events may be associated with additional information composed from the data of the events that make up the complex event.

Definition 2. Let $ce(ts) = \{event_1(ts_1, d_{11}, \ldots, d_{1m}), \ldots, event_n(ts_n, d_{n1}, \ldots, d_{nk})\}$. Then ce is a complex event defined using a set of events.

Since the occurrence of a complex events is defined by an interval, it is necessary to introduce the concept of interval.

Definition 3. For a complex event $ce(ts)$, ts is an interval defined as follows:

$$ts = [init(ce), end(ce)]$$

where:

$$init(ce) = min(\{\tau_i | e(\tau_i, d_1, \ldots, d_n) \in ce\})$$

and

$$end(ce) = max(\{\tau_e | e(\tau_e, d_1, \ldots, d_m) \in ce\})$$

and $init(ce), end(ce) \in T$.

Note that a timestamp of an atomic event can also be represented using an interval. If $ae(ts)$ is an atomic event, the interval ts is defined as $ts = [init(ae), end(ae)]$, where $init(ae) = end(ae)$.

To illustrate the need for complex events, consider the following example:

Example 2. A fire alarm has two sensors, one to measure smoke and another one to measure humidity. Both sensors are always sending their values to the alarm controller. If the controller notices that both values have reached a certain limit, it will fire the alarm. The event of firing the alarm can be seen as a complex event, because it is composed of two simple or atomic events, both from each sensor. The example can be represented in the following way:

$$fire\_alarm(ts_1, smk, hum) = \{smoke\_sensor(ts_2, smk), hum\_sensor(ts_3, hum)\}$$

where $smk \in dom(smoke\_sensor)$ and $hum \in dom(hum\_sensor)$.

Complex events are built using operators of event algebra (see section 2.1.3). This kind of algebra is very useful to describe how events can be temporally situated to other events or absolute time points. Thus, in order to combine events it is necessary to specify

the operators to combine them. These operators as well as its semantics are defined using event algebra.

2.1.3 Event Algebras

Event algebras allow to specify operators that are used to build complex events from atomic events or other complex events. These algebras have been widely discussed in the active database community. Snoop [CM94], e.g., is an active database system that uses this kind of algebra to specify the reactive behavior of the system.

Typical event algebras in active database literature, such as Snoop, define the following operators[1]:

$E_1 \vee E_2$ this operator defines the disjunction of two events and it is triggered when either event E_1 occurs or event E_2 occurs.

$E_1 \wedge E_2$ this operator defines the conjunction of two events and it is triggered only when both events, E_1 and E_2, occur at the same time.

$E_1; E_2$ the sequence of two events is triggered only when event E_2 occurs after the event E_1.

$A(E_1, E_2, E')$ this operator defines an aperiodic event. This event is triggered when the event E_2 occurs during the interval defined by the occurrence of the events E_1 and E'.

$P(E_1, t, E_1')$ this operator defines a periodic event. This event is triggered every t times during the interval defined by the events E_1 and E_1'.

According to [CL04], most of the event algebras in active databases systems treat complex events as being instantaneous, i.e., an occurrence of a complex event is associated with a single time instant, normally the time at which it can be detected. For example, the time of occurrence of the complex event $(A; B)$, defined using Snoop language, is associated with the occurrence time of B, since this is the last event needed to detect $(A; B)$.

However, as discussed in [Pas06], this approach of considering events as occurring instantaneously has some drawbacks. Consider the following example: if we define two complex events, say $A; (B; C)$ and $B; (A; C)$, it should be clear that one event should be detected if the sequence of events $< A, B, C >$ occur, and the other should be detected if the sequence of events $< B, A, C >$ occur. However, for Snoop both complex events are semantically equal, in the sense that both events occurs at the time of detection of C. This problem arises from the fact that the events, in the active database sense, are simply detected and treated as if they occur at an atomic instant. To overcome this problem, the work in [Ada05] proposed an extension for Snoop to support interval-based event specification and detection. This extension modifies the semantics of Snoop to allow

[1] The Snoop notation is adopted

Table 2.1: Allen's thirteen relationships.

Relation	Symbol	Symbol for inverse	Pictorial Example
X before Y	<	>	XXX YYY
X equal Y	=	=	XXX YYY
X meets Y	m	mi	XXXYYY
X overlaps Y	o	oi	XXX YYY
X during Y	d	di	XXX YYYYYY
X starts Y	s	si	XXX YYYYY
X finishes Y	f	fi	XXX YYYYY

composite events to be detected over an interval, instead of a single unit point. Besides extending the semantics, this work also introduces a new operator, the `Not` operator. This operator allows to describe the non-occurrence of some event during a specific interval. Besides this work overcoming the problem described above, by introducing an interval-based semantics, the semantics defined is not always clear. In particular, the semantics for the `Periodic` operator is not clear about when the event should start or terminate.

The Knowledge Representation community also proposed a way to represent and detect events using an interval-based semantics. The work proposed by [All83] defines a new algebra to describe events. In particular, Allen's interval algebra is a calculus for temporal reasoning that defines possible relations between time intervals and provides a composition table that can be used as a basis for reasoning about temporal descriptions of events. According to [All83], this interval algebra is particularly useful in domains where temporal information is imprecise and relative, and techniques such as dating are not possible. All the thirteen temporal relationships can be found in Table 2.1.

The two work, [CM94] and [All83], presented here are two distinct approaches for an event algebra. While [CM94] follows the approach of the Active Database community, which treats complex events as being instantaneous, the work in [All83] follows the Knowledge Representation community approach, in the sense that complex events are defined by an interval of occurrence.

Within this thesis context event algebra is useful to define complex events by combining simple or complex events. This complex events may be useful to model some situations that are described by the occurrence of several events.

2.1.4 Complex Event Processing

A Complex Event Processing (CEP) system is responsible for detecting complex events. These systems allow for describing events via event patterns. Event patterns describe how events can be temporally situated to other events or absolute time points.

These kinds of systems have been widely studied by the Active Database community. The Snoop language [CM94] for instance, is one of the proposed languages that are able to detect complex events. In fact, the detection of events is the main concern of the Snoop language. The detection mechanism is based on a tree corresponding to event expressions, where atomic events are inserted at the leaves and are propagated upwards in the tree in order to detect more complex events.

However, it has been recognized in several work, e.g., [AFSS09] and [EB10], that logic programming is more suitable for systems that aim at detecting events. Firstly, it has a formal declarative semantics. Secondly, integration of query processing with event processing is natural. Thirdly, a logic-based event model allows to reason over events, their relationships and possibly over some knowledge base available for a particular domain. Additionally, in an environment involving multi-threading and distributed processing, a declarative semantics guarantees predictability and repeatability, i.e., it always produces the same results when making the same query twice.

The work in [EB10] is a first approach to put event patterns in a logical framework. However, this work fails at the most important principle of the CEP systems: detection of a pattern as soon as the last event required for a complete match of the pattern has occurred. The authors' focus is only on event queries, i.e., this system is not aimed at detecting events as soon as they occur, but instead this system is only concerned to make queries to events. This means that this system can only detect an event when it poses a query. Now, one may ask: Why do not extend this work such that it poses periodic (polling) queries in order to detect events? This approach can be taken into account, but then again it fails at the most important requirement of the CEP system, besides that, it would raise performance issues. The reason that the work in [AFRSSS10] and the possible extension described above are not suitable for detecting events as soon as they occur, is that event processing relies on inference engines with the capability to do event-driven computation, i.e., based on data streams. This inability to do event-driven computation is the main drawback of most of the logic-based approaches.

Event-driven computation means that an event pattern is detected as soon as possible. According to [AFSS09], this way of detecting patterns follows the *forward chaining* approach. In contrast, most of the rule-based approaches are query-driven, i.e., for a given pattern an event processing engine needs to check if this pattern has been satisfied or not. This query-driven computation usually follows the *backward chaining* approach.

The work in [AFSS09] proposes an even-driven approach for logic-based CEP. In order to detect events it is first necessary to specify *event patterns*. An *event pattern* is a template which matches certain events. For example, an event pattern matches all orders from customers in response to a discount announcement event. This discount announcement event can be seen as an *atomic event*, which is used to build *complex events*. In general, an event pattern is a pattern which is built from events satisfying certain event operators and/or time constraints.

The authors do not use any event algebra, such as the one presented in [CL04], instead

they use Concurrent Transaction Logic (CTR) as the main underlying formalism of the logic-based approach. In short, CTR is a logic for state-changing actions. The truth of CTR formulas is determined over *paths*. A path is a finite sequence of states. If a formula, ψ, is true over a path $< s_1, \ldots, s_n >$ it means that ψ can be executed starting with state s_1. During the execution, ψ will change the current state to $s_2, s_3, \ldots$ until it reaches the terminal state s_n. Given this, the model operators can be summarized as follows:

- $\phi \otimes \psi$ means: execute ϕ, then execute ψ;
- $\phi \mid \psi$ means: execute ϕ and ψ concurrently;
- $\phi \wedge \psi$ means: ϕ and ψ must both be executed along the same path;
- $\phi \vee \psi$ means: execute ϕ or ψ nondeterministically;
- $\neg\phi$ means: execute in any way, provided that this will not be a valid execution of ϕ;
- $\odot\phi$ means: execute ϕ in an isolation of other possible concurrently running activities.

For further details about the semantics of CTR the reader is referred to [BK96].

Since CTR is a state-changing logic, the notion of an event pattern is defined as a *relevant state change* in an event-driven system, characterized by time. Thus, a logical representation of an event according to [AFSS09] is described as $e^{[T_1,T_2]}(X_1, \ldots, X_n)$, where e is an event pattern name, the $X_1, \ldots, X_n$ is a list of arguments, representing data, and $[T_1, T_2]$ is a time interval during which the event has occurred. Thus, an event pattern is defined as:

- an atomic event;
- a sequence of events: $(event_1 \otimes \ldots \otimes event_n)$, where each $event_i$ is an event pattern;
- classical conjunction: $(event_1 \wedge \ldots \wedge event_n)$, where each $event_i$ is an event pattern;
- concurrent conjunction: $(event_1 \mid \ldots \mid event_n)$, where each $event_i$ is an event pattern;
- disjunction: $(event_1 \vee \ldots \vee event_n)$, where each $event_i$ is an event pattern;
- negation: $\neg event$, where $event$ is an event pattern;

A rule is a formula of the form $eventA \leftarrow eventB$, where $eventA$ and $eventB$ are both events (atomic or complex). The following example taken from [AFSS09] exemplifies how to use event patterns:

Example 3. The complex event $checkStatus$ happens if a $priceChange$ event is followed by a $stockBuy$ event. Futher on, the two events have happened within a certain time

frame (i.e. $t < 5$). The complex event $checkStatus$ can be described as:

$$checkStatus^{[T_1,T_4]}(X,Y,Z,W) \leftarrow priceChange^{[T_1,T_2]}(X,Y) \otimes stockBuy^{[T_3,T_4]}(Z,Y,W) \otimes (T_4 - T_1 < 5).$$

However, these rules per se are not convenient to be used for *event-driven* computation, because they fail on the most important principle of the reactive systems, the ability to do event-driven computation. These rules are Prolog-style rules that can only perform query-driven computation. To overcome this problem, the authors proposed a new type of rules called *event-driven backward chaining* (EDBC), which falls into two types. The first type is used to generate goals, that represent events that have just occurred, and store those goals into the internal state. The second one checks whether a certain goal already exists in the internal state, in which case it triggers an event. Thus, any rule defined needs to be converted into EDBC rules. But before translating the Prolog-style rules into EDBC rules, event patterns need to be decomposed into intermediate events. Consider the following event pattern:

$$e^{[T_1,T_6]} \leftarrow a^{[T_1,T_2]} \otimes b^{[T_3,T_4]} \otimes c^{[T_5,T_6]} \tag{2.1}$$

This event pattern can be decomposed into:

$$e^{[T_1,T_6]} \leftarrow e_1^{[T_1,T_4]} \otimes c^{[T_5,T_6]} \tag{2.2}$$

$$e_1^{[T_1,T_6]} \leftarrow a^{[T_1,T_2]} \otimes b^{[T_3,T_4]} \tag{2.3}$$

This process is called *binarization*. According to [AFRSSS10], using the binarization is more convenient to build *event-driven* rules for three reasons. In the first place, it is easier to implement a binary event operator that considers only simple expressions, in this case atomic or complex events. Secondly, the binarization increases the possibility for sharing among events and intermediate events, when the granularity of intermediate patterns is reduced. Thirdly, the binarization eases the management of rules.

After the binarization an algorithm is executed to convert user-defined rules into EDBC rules. For example, the rule 2.3 is converted into:

$$a^{[T_1,T_2]} : -ins(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e_1[\_,\_])). \tag{2.4}$$

$$b^{[T_3,T_4]} : -goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e_1^{[T_1,T_4]}) \otimes T_2 < T_3 \otimes del(goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e_1^{[T_1,T_4]})) \otimes e_1^{[T_1,T_4]}. \tag{2.5}$$

The rule 2.4 belongs to the first type of EDBC rules and means that if the event pattern a is detected, then it inserts (**ins/1**) a goal into the database meaning that a occurred. The rule 2.5 belongs to the second type of EDBC rules and means that if the event pattern b is detected and the event a had been already detected, then it deletes (**del/1**) the goal

from the database and triggers the event pattern e_1. The algorithm used to convert user-defined rules into EDBC rules is not generic, which means that each operator has its own transformation algorithm.

ETALIS

ETALIS [AFRSSS10] is an example of a CEP logic-based language that converts user-defined rules into EDBC rules. As the work presented above, [AFSS09], ETALIS does not resort to event algebra. Instead, the language implements all the thirteen Allen's temporal relationships [All83]. Besides the thirteen relationships, ETALIS also implements other operators, such as the operators of *conjunction*, *disjuntion*, *negation* or *sequence*. For futher details about the operators, e.g., its syntax or semantics, the reader is referred to [AFRSSS10].

Thus, a complex event a which is composed of the events b and c and is triggered when the sequence of b and c occurs, can be described in ETALIS language as:

$$a \leftarrow b\ SEQ\ c$$

ETALIS[2] is implemented in Prolog. It has several modules to convert the ETALIS CEP rules into EDBC. Figure 2.1, taken from [ARFS12], shows the ETALIS architecture. It has two main components: the CEP Binarizer and the ETALIS Compiler. The first component is responsible for converting ETALIS CEP rules into binary rules. The other component is responsible for converting the binary rules into EDBC rules.

It is also possible to combine these rules with some background knowledge, which can be used to describe the domain of interest. This knowledge is also written in Prolog.

Complex event detection systems are useful to process events in order to detect event patterns, which are then used to (possibly) trigger actions. For this reason, CEP systems are relevant for this thesis, particularly the ETALIS system. This system is especially interesting because, in the first place, it is a system to detect complex events. A similar system is needed in the context of this thesis to detect complex events. Secondly, the ETALIS language is a declarative language that can be easily implemented using Prolog, which is the language used to implement the *reactive system*. And finally, ETALIS converts its user-defined rules into EDCB rules. These rules are of a special interest because they combine event-driven computation with logic-based rules. Since our reactive system will also use logic programming and will also have to monitor events in real-time, these EDBC rules are adequate for implementing our hybrid reactive system.

[2]ETALIS: https://code.google.com/p/etalis/

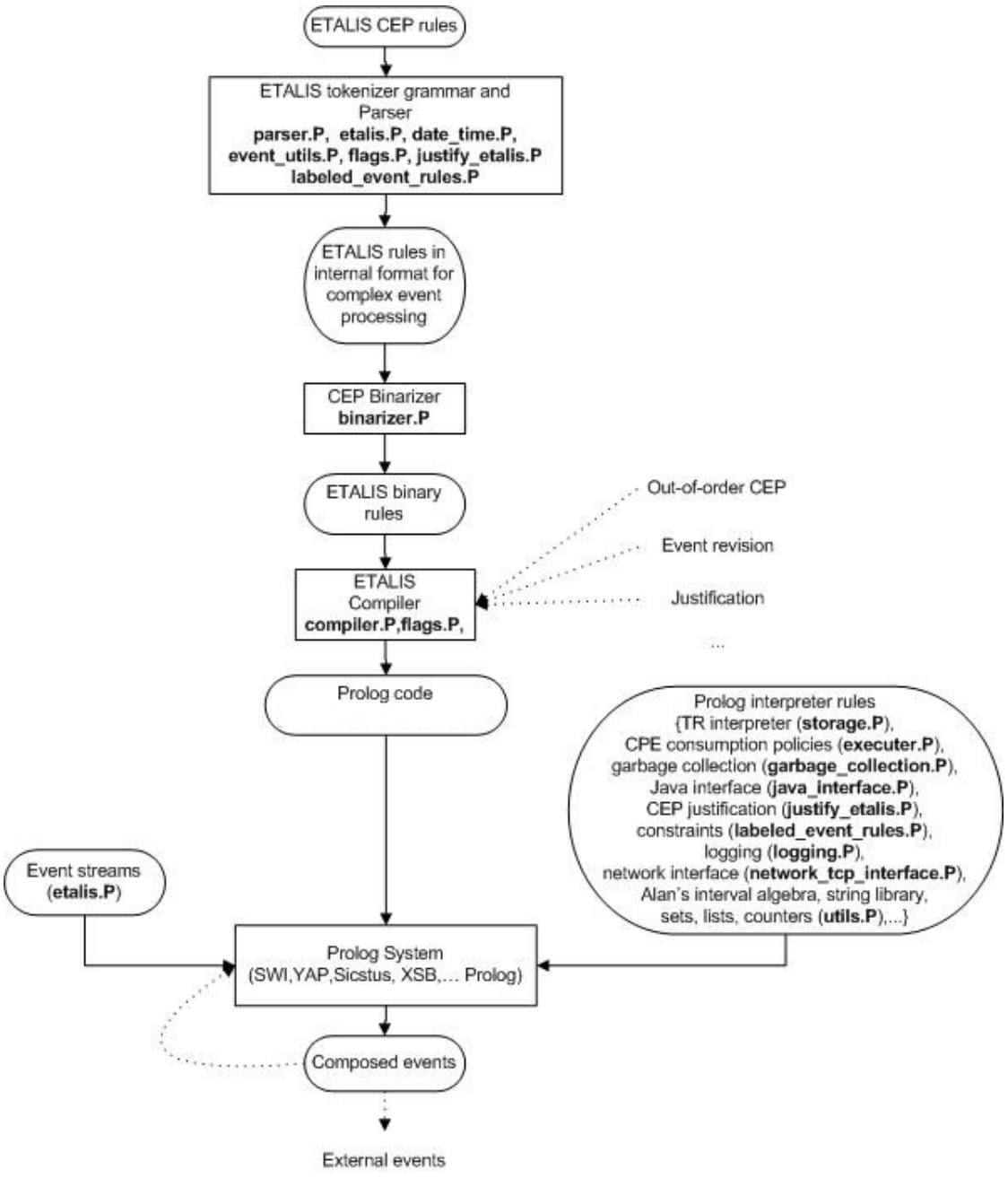


Figure 2.1: ETALIS architecture.

2.1.5 Stream Reasoning

Before explaining what *Stream Reasoning* is, it is better to start by explaining what a *data stream* is. A *Data stream* can be seen as a communication channel used to send information. In a more formal way, a *data stream* can be represented as a set of pairs (s, δ), where s is a tuple and δ is a temporal stamp. *Data streams* are used by *Data Stream Management Systems* (DSMS), such as CQL [ABW03]. These types of systems, instead of querying relations (i.e. querying data stored in tables) like most of the database systems, operate over *data streams*. The main difference is that database systems have their data stored in relations, which makes it all available at each time. On the other hand, systems based on *data streams* cannot have access to the whole data, and because of that, the queries are constructed in an incremental way. These systems are very popular, e.g., to detect frauds, monitoring networks or data market analysis. After defining *data streams*, we are now able to explain what *Stream Reasoning* is.

Stream reasoning is, by the definition in [BBCVG10],

> "logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users".

Usually, the reasoning is done over a static knowledge base in order to derive new facts. However, in stream reasoning there is no static knowledge base but instead there is a *data stream*. Giving the nature of streams, i.e., not having bounds, which makes them infinite, it is impossible for a system to perform reasoning over the entire stream. In order to make queries possible over an infinite stream, it is necessary to restrict reasoning to a window. A window is an abstraction that lets us put some bounds over a stream, i.e., instead of trying to perform reasoning over an infinite set, the reasoning is done over a subset of the stream. However, in order to always get the most recent information, it is necessary to slide the window to take into account this new information, and so to perform reasoning over the new information. A window can be either temporal or spatial. Temporal means, e.g., that the window contains all the tuples that arrived in the last 5 minutes. Spatial means, e.g., that the window contains the ten most recent tuples. For *reactive systems*, such as the one being developed in this thesis, it is important to perform reasoning tasks over some streaming data, in order to possibly detect more situations of interest.

Several work were developed in order to perform reasoning over *data streams*. The extension developed for SPARQL, Continuous-SPARQL (C-SPARQL) [BBCVG09], aims at capturing the dynamics presented today on the Web. While SPARQL relies on performing queries over RDF data repositories, C-SPARQL allows to execute queries over RDF streams. The reasoner Streaming Knowledge Bases is proposed in [WJFY08]. This reasoner deals with RDF streams and ontologies. The authors suggest to pre-compute the transitive closure with respect to an ontology and store the result in a database. Then, the reasoner can be used to identify a triple from the stream having a subject that is an

instance of a certain class defined in an ontology. This reasoner uses the system TelegraphCQ [CCDFHHKMRS03], which is a data stream processing engine capable of executing continuous queries over a stream, to store all the inferences and to perform stream reasoning.

The language proposed by [AFRS11], EP-SPARQL, is an extension of the SPARQL language with event processing and stream reasoning capabilities. The authors propose an approach to detect complex events within a stream of RDF triples. In order to detect complex events, it might be necessary to reason over background knowledge, expressed as an RDF graph or an RDFS ontology. In practice, the proposed language is implemented using the earlier mentioned system ETALIS [AFRSSS10]. The queries are mapped to *EDBC rules* and the RDF triples are mapped directly to Prolog predicates. The authors of EP-SPARQL did some performance tests to analyze the stream reasoning capabilities and the event processing evaluation. They compared their system with Esper[3], which is a state-of-the-art data stream engine (and is also used for commercial purposes), to test the performance of the event processing funcionality. The results showed that EP-SPARQL system has a better throughput than Esper. For further details about the performance test the reader is referred to [AFRS11].

2.2 Hybrid Knowledge Bases

Hybrid Knowledge Bases are knowledge bases that combine Description Logics (DL) ontologies with non-monotonic rules, thus allowing the combination of closed and open world reasoning. According to [MR10], both ontologies and rules exhibit certain shortcomings that can be compensated by the features of the other formalism. For example, as mentioned in [KAH11], rules make it possible to express: non-treeshape-like relationships, such as "an uncle is the brother of one's father"; integrity constraints to state, e.g., that a certain piece of information is explicity present in the database; and closed world reasoning. On the other hand, ontologies make it possible to express open world reasoning, reasoning with infinite domains and thus are suitable to represent many types of incomplete information. Thus, complex knowledge representation problems often require features found in both DLs and rules.

The authors of [MR10] argue that a formalism combining ontologies and rules should satisfy the following criteria:

Faithfulness The integration of DLs and rules should preserve the semantics of both formalisms - that is, the semantics of a hybrid knowledge base in which one component is empty should be the same as the semantics of the other component. In other words, the addition of rules to a DL should not change the semantics of the DL and vice versa.

[3] http://esper.codehaus.org

Tightness Rules should not be layered on top of a DL or vice versa; rather, the integration between a DL and rules should be tight in the sense that both the DL and the rule component should be able to contribute to the consequences of the other component.

Flexibility The hybrid formalism should be flexible and allow one to view the same predicate under both open and closed world interpretation. This allows us to enrich a DL with nonmonotonic consequences from rules, and to enrich the rules with the capabilities of ontology reasoning described by a DL.

Decidability To obtain a useful formalism that can be used in applications such as the Semantic Web, the hybrid formalism should be at least decidable, and preferably of low worst-case complexity.

The combination of DL ontologies and rules have been a constant challenge for the Knowledge Representation community, in the sense that a combination of DL with first-order rules easily leads to undecidability of the basic reasoning problems even if both the DL and the rule formalisms alone are decidable.

Among several proposals for combining both formalisms (e.g [DM07], [EILST08] and [MR10]), the only one that satisfy the above criteria is the one proposed in [MR10]. This work proposes a formalism of $MKNF^+$ knowledge bases, which allows for a faithful, tight, and flexible integration of DLs and answer set programming (ASP). The formalism defined in this work is based on the logic of minimal knowledge and negation as failure (MKNF [Lif91]), which goal is to unify most existing approaches to nonmonotonic reasoning, and is the one adopted by the ERRO team to combine ontologies with nonmonotonic rules, with the difference that in ERRO project the semantic used is the one defined in [KAH11].

To illustrate the usage of $MKNF^+$ knowledge base, consider the following simplified version of a MKNF Knowledge Base about Cities, taken from [MR10]:

$$portCity(Barcelona) \tag{2.6}$$

$$\neg seasideCity(Hamburg) \tag{2.7}$$

$$seaside \sqsubseteq \exists has.beach \tag{2.8}$$

$$K\ seasideCity(x) \leftarrow K\ portCity(x), not\ \neg seasideCity(x) \tag{2.9}$$

$$K\ HasOnSea(x) \leftarrow K\ onSea(x, y) \tag{2.10}$$

$$false \leftarrow K\ seasideCity(x), not\ HasOnSea(x) \tag{2.11}$$

The DL part consists of the axioms 2.6 - 2.8 that state that Bacelona is a port city (2.6), Hamburg is not a seaside city (2.7) and that seaside cities have a beach (2.8). The rules part consists of the rules 2.9 - 2.11 and state that port cities are usually at the seaside (2.9) and that for each seaside city we must know on which sea the city is (2.10 and 2.11). For example, Barcelona is a port city by 2.6, and there is no evidence that it is not a seaside

city, so we derive $seasideCity(Barcelona)$. In contrast, 2.7 explicity says that Hamburg is not a seaside city. Hence, Hamburg is an exception to rule 2.9.

2.3 ECA Rules

Event-Condition-Action rules, as mentioned before, have the following format:

On event **If** condition **Then** action

Reactive systems are described using these rules so they can be able to react and evolve over some background knowledge. In the active database systems this background knowledge is just a set of tuples stored in relations. With the emergence of the Semantic Web this knowledge can be either a simple set of facts or a much richer knowledge base containing more than simple facts, allowing for the specification of both relational data and classical rules, i.e., rules that specify knowledge about the environment. However, as mentioned before, combining ECA rules with richer knowledge bases have become a challenge.

There are some work towards that direction, e.g., [BPW02], [PPW04] and [AEM09]. Knowing that XML has becoming a dominant standard for storing and exchanging information on the Web, the work in [BPW02] proposed an ECA language for XML repositories. The goal of this work is to provide a language to support the reactivity funcionality on XML repositories. Events are triggered when some modifications occurs to the repository. As a response to the event triggering, actions are executed over the repository. The ECA rules are described using XPath and XQuery languages. XPath is used for selecting and matching XML documents within event and condition parts. XQuery is used on action parts to construct new XML fragments.

In [PPW04], the authors also proposed an ECA rule language, the RDF Triggering Language. The RDFTL language provides reactivity functionality over RDF metadata stored in RDF repositories. The authors have chosen RDF because it is becoming a core technology in Semantic Web and it provides a way for describing metadata information that can be easily navigated. It uses path expressions to query RDF metadata. The event part of a rule detects when resources or arcs specified by triples are inserted or deleted on the RDF triples/graph. It also detects when arcs are updated. The action part allows for insertions and deletions of resources and arcs and also allows for update arcs.

The work in [AEM09] proposes a general framework for reactive ECA rules in Semantic Web and also a concrete homogeneous language, XChange. Given that the Semantic Web does not have a centralized structure, the communication is based on peer-to-peer nodes. This means that each node will have different data models and languages to represent evolution and behaviour on the Semantic Web. The general framework was developed so that evolution and reactivity on the Semantic Web could be based on a general ECA language that allows for the usage of different languages for events, conditions and actions.

2.4 Logic-based Languages and Tools

2.4.1 RIF

The Rule Interchange Format (RIF) is a standard for exchanging rules amongst rule systems, in particular among Web rule engines.

The existing rule systems fall into three main rule categories: first-order, logic programming and action rules. Even within each paradigm there are several differences between systems. Thus, given this diversity, the main focus of the Working Group has been on exchanging rather than trying to create a general language for all the existing systems.

The Working Group decided to create a family of languages, called dialects. This dialects were made to be uniform and extensible. Uniform so they can share as much as possible the syntax and semantics, and extensible to allow this dialects to be extended with new features.

The RIF Working Group focused on two kinds of dialects: **1) logic-based dialects** include languages that allow some kind of logic, e.g., first-order logic. **2) dialects for rules with actions** include production rules, e.g., Jess[4] or Drools[5], and reactive rules, e.g., Reaction RuleML[6] or XChange[7].

Until now, the Working Group developed three dialects:

RIF-BLD Basic Logic Dialect [BK13] corresponds to Horn rules without function symbols.

RIF-PRD Production Rule Dialect [SMHP13] captures the mains aspects of production rule systems.

RIF-Core Core Dialect [BKHPPR13] is a subset of RIF-BLD and RIF-PRD which enables a limited rule exchange among logic-based dialects and dialects for rules with actions.

RIF-PRD deals with production rules, which are a particular type of *reactive rules*. Production rules can be seen as condition-action rules. According to RDF-PRD, the condition part of the rules is like the condition part of logic rules, whereas the action part can assert facts, modify facts, retract facts and have other side-effects (as opposed to conclusion in logic rules).

The following rules are examples of production rules taken from [SMHP13].

1. A customer becomes a "Gold" customer when his cumulative purchases during the current year reach $5000.

[4] http://herzberg.ca.sandia.gov/
[5] http://www.jboss.org/drools/
[6] http://ruleml.org/reaction/
[7] http://reactiveweb.org/xchange/

2. Customers that become "Gold" customers must be notified immediately, and a golden customer card will be printed and sent to them within one week.

3. For shopping carts worth more than $1000, "Gold" customers receive an additional discount of 10% of the total amount.

RIF-PRD specifies an abstract syntax[8] that shares features with concrete production rules languages, and associates it with normative semantics and normative XML concrete syntax.

For instance, the first rule can be represented as [SMHP13]:

```
Prefix(ex <http://example.com/2008/prd1#>)
(* ex:rule_1 *)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

The RIF-PRD operational semantics for production rules are summarized as follows:

Match: the rules are instantiated based on the definition of the rule conditions and the current state of the data source.

Conflict resolution: a decision algorithm, often called *the conflict resolution strategy*, is applied to select which rule instance will be executed;

Act: the state of the data source is changed, by executing the selected rule instance's actions. If a terminal state has not been reached, the control loops back to the first step (**Match**).

The work in [DAL10] introduces a declarative logical characterization of the full default semantics of RIF-PRD based on Answer Set Programming (ASP), including matching, conflict resolution and acting. This specification uses logic programming semantics to encode the effects and dynamics of knowledge base change, which facilitates the integration with ontologies, namely with Hybrid Knowledge Bases.

2.4.2 XSB-Prolog

XSB-Prolog is a Prolog extension with features of deductive databases and non-monotonic reasoning. It provides nearly all functionality of ISO-Prolog and other important features, as mentioned in [SW13], such as *tabling*, *multi-threading programming* and *trie storage*.

[8] The abstract syntatic constructs are defined in:
http://www.w3.org/TR/2013/REC-rif-prd-20130205/#sec-conditions-abstract-syntax
http://www.w3.org/TR/2013/REC-rif-prd-20130205/#sec-actions-abstract-syntax
http://www.w3.org/TR/2013/REC-rif-prd-20130205/#sec-rules-abstract-syntax

Tabling is the most important feature of XSB and it is what differentiates XSB from other Prolog systems. The *tabling* is a "simple" feature yet has huge implications. To demonstrate the use of *tabling* consider the following example taken from [SW13]:

```
:- table ancestor/2.

ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

Any query to the above program will terminate in XSB, since `ancestor/2` is compiled as a tabled predicate. However, any other Prolog system would go into an infinite loop. This happens because Prolog is based on a depth-first search through trees that are built using program clause resolution (SLD), which may lead to the Prolog system getting lost in an infinite branch of a search tree. However, XSB allows the user to use SLG [CW96] resolution to evaluate such logic programs. SLG is a table-oriented resolution method that combine deductive databases, non-monotonic reasoning and logic programming paradigms. SLG resembles SLD in that it admits a tuple-at-a-time resolution method, so it can make use of many of the techniques developed in implementing SLD. SLG resolution can be declared using the `table` declaration. It is also possible to let the system decide which predicates need to be tabled, using the `auto_table` declaration.

Prolog was designed to be fully declarative, which would allow programmers to simply describe the problems. However, in practice, this is not what happens. When writing Prolog programs, programmers can not place the clauses in some random order, because Prolog has a very specific way of working out the answers to queries. If some clauses are placed randomly, the Prolog program can enter into a loop. The *tabling* feature presented here is an important feature because it allows Prolog programs to be more declarative, and thus allowing for programmers to place the clauses without worrying about possible loops and reasoning algorithms.

Multi-threading programming was introduced with version 3.0, and allows XSB programmers to use POSIX threads to perform separable computations, and in certain cases to parallelize them. The programmer can use the available message queues to make the threads communicate with each other using a queue. These queues make a consumer to suspend its work until some goal G unifies with a predefined term T.

The *trie storage* is a mechanism by which large numbers of facts can be directly stored and manipulated using tree data structures (*tries*). When stored in a trie, facts are compiled into trie-instructions similar to those used for XSB's tables. For instance, the set of facts taken from [CW96]

$$\{rt(a, f(a, b), a),\ rt(a, f(a, X), Y),\ rt(b, V, d)\}$$

would be stored in a trie as shown in Figure 2.2. According to [CW96], using a trie for storage is 4-5x faster than with standard dynamic code.

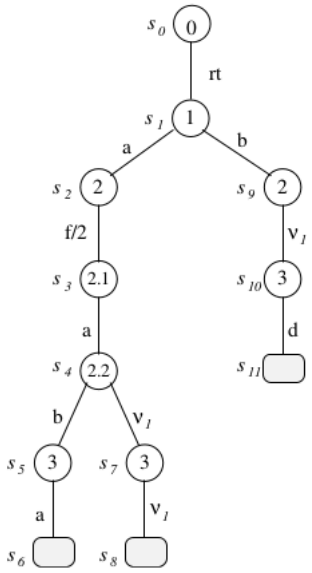


Figure 2.2: Terms stored as a trie.

2.4.3 NoHR Plugin

NoHR (Nova Hybrid Reasoner) is a Protegé plug-in that was developed within the ERRO project context. According to [IKL13], the plug-in allows the user to take an $\mathcal{EL}^+_\bot$ ontology, add a set of non-monotonic rules and query the combined knowledge base. The tool builds on the procedure *SLG(O)* [AKS10] and pre-processes the ontology into rules, whose result together with the non-monotonic rules serve as input for the XSB Prolog engine. The system is able to deal with (possible) inconsistencies between rules and an ontology that alone is consistent. The architecture of the NoHR plugin is depicted in Figure 2.3, taken from [IKL13].

2.4.4 OWL API

The OWL API [HB11] is a high level Application Programming Interface (API) for working with OWL ontologies. It supports parsing and rendering in the syntaxes defined in the W3C[9] specification (e.g., RDF/XML or OWL/XML), manipulation of ontological structures and the use of reasoning engines. OWL API, which is written in Java[10], provides several abstractions to deal with ontologies, i.e., it provides an interface for accessing the axioms contained in an ontology (`OWLOntology`) and also provides a central point for creating, loading, changing and saving ontologies (`OWLOntologyManager` interface). However, OWL API has some limitations, in the sense that the OWL API is primarily designed to be an application that supports manipulation of OWL ontologies at a particular level of abstraction - which is not the RDF level - making it less suitable

[9] http://www.w3.org/
[10] https://www.java.com/en/

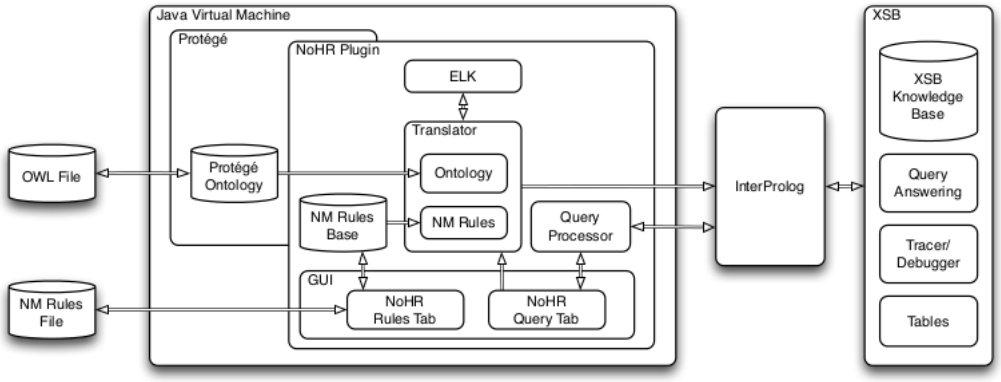


Figure 2.3: NoHR architecture

for those wishing to exploit the layering of OWL on RDF.

2.4.5 Protégé-OWL API

The Protégé-OWL API[11] is an open-source Java library for the Web Ontology Language and RDF(S). The API provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning. Furthermore, the API is optimized for the implementation of graphical user interfaces.

The API is designed to be used in two contexts:

- For the development of components that are executed inside of the Protege-OWL user interface editor
- For the development of stand-alone applications (e.g., Swing applications, Servlets, or Eclipse plug-ins)

2.4.6 ELK Reasoner

According to [KKS12], ELK is a reasoner that provides high performance reasoning support for OWL EL ontologies. The main focus of the system is the extensive coverage of the OWL EL features, the high performance of reasoning and the easy extensibility and use. Today, ELK is the only system that can use multiple processors to speed up the reasoning process, making it possible to classify SNOMED CT ontology[12] in about 5 seconds. Since version 0.4.0, ELK is capable of incrementally updating the inferred class and instance hierarchies after modifying axioms. The supported types of axioms that can be changed incrementally are: *SubClassOf*, *EquivalentClasses*, *DisjointClasses*, *ObjectPropertyDomain*, *ClassAssertion* and *ObjectPropertyAssertion*. However, there are some axioms that changing them affects the property hierarchy or property chains and will still trigger

[11] http://protege.stanford.edu/plugins/owl/api
[12] http://www.ihtsdo.org/snomed-ct/

the full re-classification. The authors refers that these changes are typically non-local, so incorporating them incrementally does not pay off. Those axioms are: *SubObjectPropertyOf*, *SubDataPropertyOf*, *EquivalentObjectProperties*, *EquivalentDataProperty*, *TransitiveObjectProperty* and *ReflexiveObjectProperty*.

2.5 Conclusion

In this section we introduced several approaches that will serve as a base for our hybrid reactive system.

In the first section (2.1) we introduced several work that deals with events. In particular, we presented the ETALIS language, which is a logic-based approach to detect complex events. This provides a good basis for our hybrid reactive system, in the sense that it explains how complex events can be detected in real time using logic programming, which will be the paradigm used in our work.

Next, we presented some works related to reactive rules for the Semantic Web. These work are simply concerned on acting over repositories of data when some data there present is modified. Our work goes much beyond that, in the sense that our reactive system will act over much richer knowledge bases, the so-called hybrid knowledge bases, introduced in Section 2.2.

We finished this chapter presenting logic-based rule technologies. We described several tools that will be used to implement our system, such as the XSB Prolog. Some other technologies, such as RIF-PRD, will not be used in our work, but will be the base to implement our language for exchanging ECA rules amongst Web rule engines.

3

Architecture

In this chapter, we are going to briefly overview the whole *reactive system*. Here, the final architecture of the system is presented aiming to familiarize the reader with it to improve the understanding of the forthcoming chapters. This chapter is divided into two main sections, one about the *reactive system* itself and one about the integration with *Protégé*. The explanation is not overly detailed, instead the technical explanations and decisions are left to the subsequent chapters.

3.1 Reactive System

One of the goals of this thesis is to create a system capable of reacting to the detection of events of interest with respect to a hybrid knowledge base. To meet that goal, we decided to rely on ECA rules as the basis for the reactivity part of our system. As discussed earlier, these rules are well-suited for the reactive, event-based aspect of Web applications. Despite ECA rules allowing the detection of events, they cannot be used to perform temporal reasoning over the event stream, meaning that many situations of interest will not be captured by the reactive system.

As shown in the previous chapter, event algebras are useful to combine atomic/complex events, through the use of proper operators, in order to describe some situations of interest related with the occurrence of events. Using these operators, the user can declare *patterns* in order to define new events, also called complex events. However, complex event processing systems alone, such as ETALIS, are not so interesting in situations where there is need for reactivity, because their goal is just to trigger atomic or complex events, delegating the event processing to other systems.

The system developed within the context of this thesis can fill the gap between the

ECA rules and the complex event rules, in the sense that it combines both kinds of rules in order to make them work in a cooperative way, making it a *reactive system* with capabilities to do temporal reasoning. None of the systems presented in Chapter 2 can combine these two technologies (ECA rules and rules with complex events) into a single framework as our system does. Besides combining these two technologies, our system is developed using a logic programming paradigm. Given the declarative nature of such paradigm, a system for supporting ECA rules can benefit from such paradigm, in the sense that the ECA rules can be easily described in a logical system, making those descriptions very close to natural language. The query mechanism from these rules can also take advantage of the *query-driven* capability that most logical systems have, such as XSB Prolog.

In order to obtain a very expressive language that allows one to describe as many situations as possible, our system combines Allen's interval algebra with the SNOOP event algebra operators. The resulting definition of complex events can then be used on the event part of the ECA rules, as if those events were atomic.

From the user's point of view, the system requires that the user describes the complex event rules and the ECA rules and provide those rules as an input.

Besides providing the ECA and complex event rules, the user can encode the domain of interest, i.e. the domain where the system is being used, in the form of a set of facts and rules. The former are predicate expressions that make a declarative statement about the problem domain, and the latter are predicate expressions that use the logical implication to describe a relationship among facts. Those set of facts and rules compose the background knowledge of our system, where actions will be performed in order to make that knowledge evolve.

Actions are declared in the action part of the ECA rules, and they can be one of the following: an *insertion* or a *removal*. The content of an action can be either a fact or a rule.

For the system to work, it has to receive events in order to trigger other events or to perform actions over the background knowledge. The events can be sent to the system through an event stream or sent one by one. The event stream is a set of events, written in a file, that are passed to the system to be executed automatically one by one. Alternatively, if the user prefers, the events can be sent manually one by one.

Whenever an action is performed, our system creates a new event and sends it to itself indicating that an action was triggered. Each action has its own event name, which can be used by the user when defining complex events or the event part of an ECA rule. These events only contain the content of the action performed, i.e., either a fact or a rule.

Whenever an event is triggered, being it atomic or complex or an event describing a triggered action, the user is notified by the system, displaying the event on the screen.

The workflow of the reactive system along with a screenshot of the notifications sent by it are depicted in Figures 3.1 and 3.2, respectively.

Figure 3.1: Workflow of the reactive system

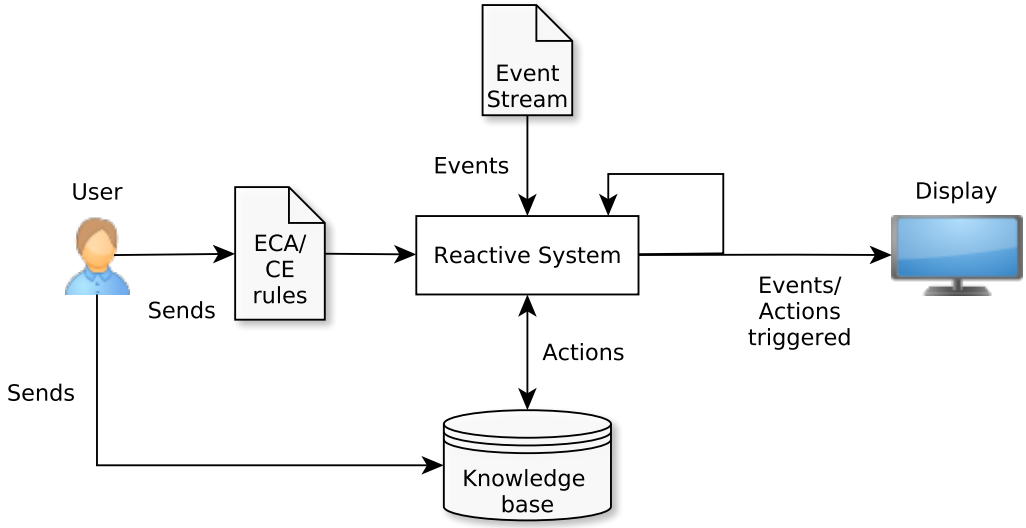


Figure 3.2: Reactive system notifying the user

```
| ?- [etalis loaded]
[logging loaded]
[binarizer loaded]
[compiler loaded]
[executor loaded]
[flags loaded]
[labeled_event_rules loaded]
[operators loaded, cpu time used: 0.0080 seconds]
[parser loaded]
[storage loaded]
[utils loaded]
[periodic loaded]
[p_queue loaded]
[kb_interface loaded]
[java_communication loaded]
*Event: a(2) @ [datime(2014,8,25,10,38,4,1),datime(2014,8,25,10,38,4,1)]
               datime(2014,8,25,10,38,4)
*Event: b(3) @ [datime(2014,8,25,10,38,4,2),datime(2014,8,25,10,38,4,2)]
               datime(2014,8,25,10,38,4)
*Event: a(3) @ [datime(2014,8,25,10,38,4,1),datime(2014,8,25,10,38,4,2)]
               datime(2014,8,25,10,38,4)
*Event: a(1) @ [datime(2014,8,25,10,38,4,3),datime(2014,8,25,10,38,4,3)]
               datime(2014,8,25,10,38,4)
*Event: b(2) @ [datime(2014,8,25,10,38,4,4),datime(2014,8,25,10,38,4,4)]
               datime(2014,8,25,10,38,4)
*Event: a(2) @ [datime(2014,8,25,10,38,4,3),datime(2014,8,25,10,38,4,4)]
               datime(2014,8,25,10,38,4)

yes
| ?-
| ?-
```

3.2 Protégé Plug-in

So far, we briefly described the reactive system and how it works. In this section, our focus will be on the other main contribution of this thesis, the Protégé plug-in.

As explained in the previous section, the reactive system was developed to be used as an extension of XSB Prolog, where a user can construct ECA rules and complex event rules to be triggered upon the arrival of events. However, this system is not prepared to be used in an environment such as the Semantic Web, where ontologies, for example, are frequently used. Namely, ECA rules and complex event rules cannot be described in a uniform format so that they can be shared among the different engines on the Semantic Web.

As already mentioned in Chapter 1, Protégé is an open-source ontology editor and framework for building knowledge-based solutions. This framework is ideal not only for those who want to manage ontologies but also for those who want to create new technologies for the Semantic Web, that depend on ontologies.

Having this in mind, and knowing the importance of the Semantic Web nowadays, we took advantage of the tools provided by the Protégé framework and developed a plug-in aiming to expose the reactive system to the Semantic Web environment. Indeed, the goal of our work begin developed in the context of this thesis is not only to use ontologies, but instead combine ontologies with non-monotonic rules.

One of the most important features of our system is indeed the capability to handle so-called *hybrid knowledge bases* that combine ontologies with the non-monotonic rules. These knowledge bases constitute the background knowledge of our system. To combine ontologies with non-monotonic rules, NoHR uses a translation to rules. The resulting rules, which are written in XSB Prolog syntax, form the hybrid knowledge base. Since the rules are written in XSB Prolog syntax, they are compatible with our reactive system.

This plug-in becomes now the bridge between the reactive system and the user. With this solution the user has no longer to provide the knowledge base as a set of facts and rules, but instead it is asked to provide an ontology and/or a set of non-monotonic rules to form the knowledge base.

For the rules related with the events (ECA rules and complex event rules), they can now be described using our dialect, besides the Prolog similar syntax. This dialect was developed to fill the gap left by the W3C consortium when it comes to reactive rules, since they only provided the dialect for the production rules.

Whenever the user decides to use the dialect, the plug-in is accompanied with an Extensible Stylesheet Language (XSL) file that is capable of converting the rules written using the dialect into the syntax supported by the *reactive system*.

The plug-in has a graphical interface, which can be seen in Figures 3.3 and 3.4, where the user can interact with the reactive system. The interface is divided into two views: the *files editor tab* and the *event tab*. The first one is used to load and edit the files containing the non-monotonic rules and the rules related with the events. These files are the ones given

as input for the reactive system. The second one is where the system can be initialized and is where the user can send the events to the *reactive system*. It is also possible to visualize the progress of the system through its output messages.

The ontology is neither loaded or visualized in the first view, because Protégé already has tabs for that purpose.

When the system is initialized it creates the rules file corresponding to the hybrid knowledge base, sends it along with the event rules to the *reactive system* and then waits for the arrival of the events. The events can be sent one-by-one by the user on the *Events tab* or, before initializing the system, the user can load a file on the *Files editor tab* with a list of events to be executed. This last alternative to send events is just a way for the user to avoid sending the events one-by-one, making the process more convenient.

Whenever an action occurs, it means that some modification is made to the background knowledge. In the system described in the previous section, only facts and rules could be inserted or retracted from the knowledge base. Now, since a hybrid knowledge base is being used as background knowledge, not only facts and rules can be inserted or retracted but also axioms. This means that, in the context of our solution for the plug-in, a distinction is made between the actions to be performed over the ontology and the actions to be performed over the non-monotonic rules.

Summing up, the work flow of the whole solution is depicted in Figure 3.2. The user loads the ontology file in Protégé and loads the non-monotonic rules file and the file containing both ECA rules and complex event rules in the *Files editor tab*. The Protégé plug-in then combines the ontology with the non-monotonic rules resulting in the hybrid knowledge base.

Whenever the plug-in receives an event, it sends it to the reactive system, looking up for complex events or actions to be triggered. When it happens, the reactive system applies the action to the hybrid knowledge base and notifies the plug-in indicating the occurred action.

In conclusion, our solution can cover a large spectrum of applications, from medicine to e-commerce. With our system developed using the XSB engine, we made available a technology that combines reactive rules, in particular ECA rules, with complex event processing, which is used for detecting complex events, along with the ability to interact with a background knowledge base.

To make such a reactive system available to be used in the so called Web 3.0, we also developed a Protégé plug-in that is capable of combining ontologies with non-monotonic rules, producing a hybrid knowledge base, and also takes advantage of the reactive part of the previous engine to make that knowledge evolve. A new dialect was also proposed to exchange the event rules amongst the Web engines.

As far as we know, there are no such systems available.

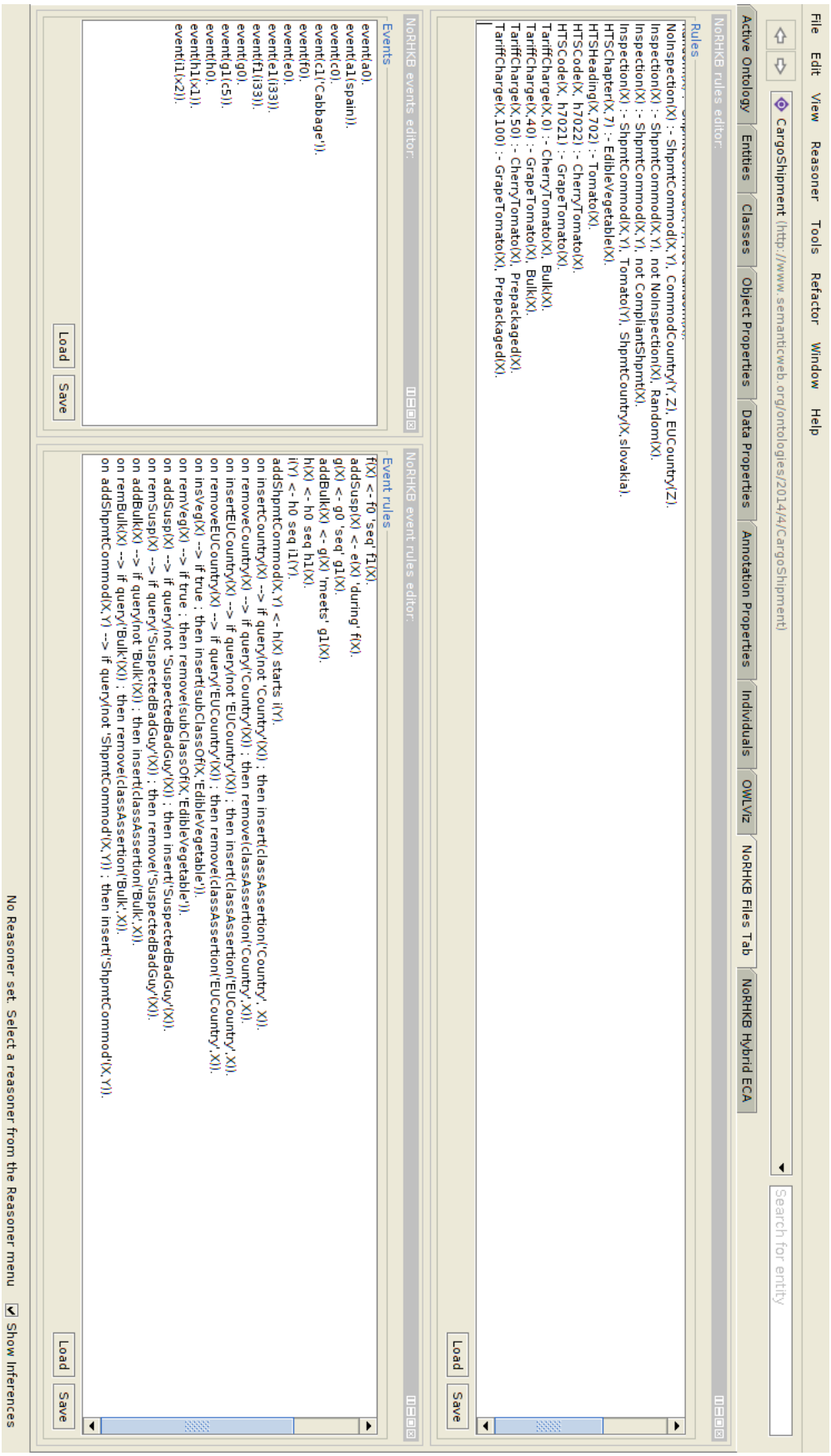
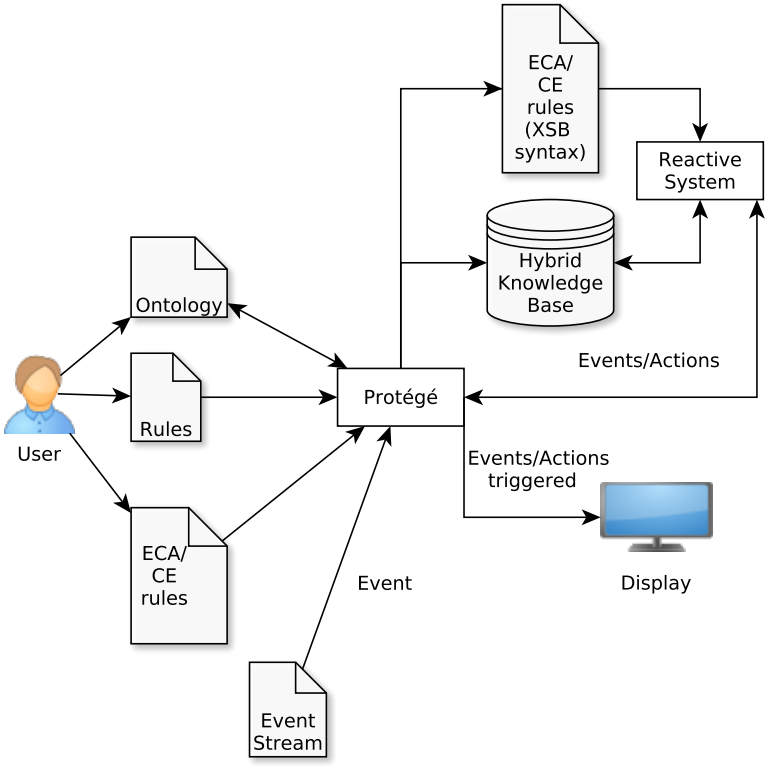


Figure 3.3: Files editor tab

Figure 3.4: Event tab

Figure 3.5: Complete solution workflow



4

Supported Language

One of the contributions of this thesis is a RIF-based language for exchanging rules among the Web rule engines. In Section 2.4.1, we introduced RIF and, in particularly, we introduced the RIF-PRD dialect. This dialect is useful to exchange production rules, which is one type of reactive rules. However, for the other type of reactive rules, the Event-Condition-Action (ECA) rules, there is no such dialect, leaving these rules without a uniform format to be exchanged on the Web.

Since we are developing a reactive system that relies on this kind of rules, we informally introduce here our proposal for a new dialect for ECA rules. The technical description for this dialect can be found in Appendix A.

Since ECA rules are a logical extension of the production rules, i.e., production rules are ECA rules without the event part, our approach is to use the RIF-PRD dialect as the base dialect for the reactive rules dialect. However, we are not only interested in ECA rules, but also interested in being able to express complex event rules. Thus, our goal is to augment the base dialect with the event part and the complex event rules declaration.

ECA rules have an *event* part, a *condition* part and an *action* part. The *event* part allows one to declare which event will trigger the corresponding ECA rule. The *condition* part is used as a guard that must be satisfied before the action of the ECA rule is executed. Finally, the *action* part is where one can declare the action to be performed over some background knowledge.

To better illustrate the need for this kind of rules, consider the following example:

Example 4. Whenever some warning about an employee is raised and that employee is a new employee, then he/she should be put in some watch list. Using the ECA rules, this

situation can be described as follows:

```
Prefix(ex <http://example.com/eca#>)
ECAGroup(
On ex:employee_warn(?E)
If (And(ex:employee(?E) ex:status(?E "new")))
Then Do(Assert(ex:watchlist(?E)))
)
```

This example shows us a simple situation where ECA rules can be used. For the sake of readability, a presentation syntax is used instead of the XML syntax.

In this example, we can easily identify each part that composes the ECA rules. The `On ex:employee_warn(?E)` part represents the event that will make this ECA rule trigger. The event is accompanied with additional information (`?E`) that will be used to refer to an employee.

The *condition* is represented by `If (And(ex:employee(?E) ex:status(?E "new")))`. This is the guard that must be satisfied before an action is executed. The guard is important to guarantee that the actions are executed only in certain situations. In this particular case, the action is only executed when the employee is a new employee.

Finally, the `Then Do(Assert(ex:watchlist(?E)))` part represents the action that will be executed over some background knowledge. In this case, the action will insert the employee on some watch list.

A set of ECA rules can be encapsulated within a group (`ECAGroup`). This notion of group was already introduced in RIF-PRD and is useful when conflict resolution strategies are used to give priority to a group of ECA rules over some other group of ECA rules.

When it comes to ECA rules, these are somehow limited in the type of events that they can detect. These rules can only detect atomic events, meaning that there is no way to compose events to also detect complex situations. With this in mind, we also provide support for complex event rules in this dialect.

Complex event rules are useful to capture certain situations of interest such as the one described in the next example:

Example 5. When some employee leaves the work before the working day ends, an event should be raised. This situation can be described as follows:

```
EGroup(
ex:employee_warn(?E) <- ex:employee_left(?E) during ex:working_day()
)
```

This example demonstrates how a complex event (`ex:employee_warn(?E)`) can be described by combining other events (`ex:employee_left(?E)` and `ex:working_day()`). These events are combined using a particular algebra, called event algebra. The `during`

operator is one of the available operators. This dialect has full support for Allen's interval algebra and for the SNOOP algebra.

For the same reason as explained before, the complex event rules can also be grouped (`EGroup`).

Summing up, the language informally presented here is based on RIF-PRD mostly due to the similarity between the two types of reactive rules.

Production rules do not deal with events, meaning that a vocabulary is missing to deal with events in RIF-PRD. Thus, the event part of the ECA rules served as starting point to develop our language. In this process, we aimed at following the same structure presented in the RID-PRD document, which can be witnessed, e.g., by the fact that our language is also divided into subsets of languages - one for each of its constituent parts.

5

Reactive System

The reactive system has the ability to detect complex events and also to trigger actions upon the detection of some event of interest, making it possible to a user to describe complex events and ECA rules. Since one of the main objectives is the integration with a hybrid knowledge base, which is translated into XSB Prolog, we designed our system using the logical programming paradigm to favor the communication with the hybrid knowledge base. Besides that, as mentioned in Section 2.1.4, the logic paradigm has several advantages over other paradigms when it comes to detecting complex events.

In Chapter 3 we briefly describe this reactive system. Throughout this chapter, our focus will be on its implementation. This chapter has two main sections: **complex event processing** and **ECA rules processing**. The first section is dedicated to describe in detail how the system was implemented in order to be able to detect complex events. The second section is used to discuss how the complex event system was augmented so it can also react upon the events arrival.

5.1 Complex Event Processing

Taking into account the state-of-the-art of complex event processing systems, our first concern was to decide whether to start the implementation from scratch or to build on an already developed system. Since we were interested in a logical way to detect complex events, our choice was to use the ETALIS project as a starting point.

As presented in Section 2.1.4, the ETALIS team developed a logic-based system that is able to detect complex events. To achieve that, they developed a new kind of rules (already discussed in Section 2.1.4) called Event-Driven Backward Chaining rules or EDBC rules. With these new rules, it is possible to have event-driven computation in a system

that is based on backward chaining computation. These rules were not developed to be used by the common user, but instead to be used internally by the system. For the common user a user-friendly syntax that is very close to that of Prolog was provided.

Due to the characteristics of the ETALIS system and knowing that the code is freely available to use, it was a natural choice to use the ETALIS as our basis.

The first step towards the implementation of the logical reactive system was to take the ETALIS source code and start studying their implementation, to better understand how it was developed, and then remove all the pieces of code that will not be used in our implementation.

The first difficulty arose in this initial phase. The ETALIS system is composed of several modules and most of the modules contain a significant amount of code. For instance, the **compiler** module had almost 2000 lines of code. Besides having a considerable amount of code, the implementation was more complex than what appeared in the paper describing it and it lacked a good documentation explaining the most critical pieces of code. In Figure 2.1, the modules that compose the original ETALIS system are described.

The result of this first phase was the elimination of some unwanted operators, the memory management module, the Java communication module, the network communication module and pieces of code related with the retraction of events, consumption policies and code not related with XSB. The reason to remove them is because some of these modules did not have support for XSB or was not in our plans to keep such features in our system.

As mentioned in the ETALIS paper [AFRSSS10], the ETALIS team wanted to cover as much Prolog engines as possible. In fact, they state that ETALIS was tested in YAP,[1] SWI,[2] SICStus,[3] XSB,[4] Tuprolog,[5] and LAP Prolog.[6] Since most of the Prolog engines available follow the Prolog standard, their syntax is compatible in most of the cases, however, for certain non-trivial features, such as the memory management, the ETALIS team only provided support for YAP Prolog and SWI Prolog, leaving the other engines without support for these specific features.

To keep full compatibility with all these engines, the ETALIS system cannot take advantage of several particular features available in some engines, such as the XSB tabling feature.

During the cleaning phase, we observed that the *findall* predicate was used to retrieve all the instances, for a specific predicate, stored internally. Knowing how most Prolog systems work, the use of the *findall* predicate is not very efficient when the facts are stored using the standard storage mechanism. Having this in mind, several tests were run in

[1] http://www.dcc.fc.up.pt/~vsc/Yap/
[2] http://www.swi-prolog.org/
[3] https://www.sics.se/projects/sicstus-prolog-leading-prolog-technology
[4] http://xsb.sourceforge.net/
[5] http://apice.unibo.it/xwiki/bin/view/Tuprolog/
[6] http://www.lpa.co.uk/win.htm

order to test how the system execution time was distributed. The *profiling* mechanism available in XSB helped us to observe that indeed, the system was wasting most of its execution time collecting facts through the use of the *findall* predicate.

This observation led us to do another significant modification in comparison to the original ETALIS system in the way it stored its facts. The standard storage mechanism used was replaced by the trie storage mechanism available only in XSB, which is, according to [SW13], 4-5 times faster than the standard storage mechanism.

Making this considerable change, we expect the system to reduce its execution time compared with the original implementation. To verify that, some tests were conducted and the results are presented and discussed in Chapter 7.

Achieving this improvement was non-trivial given the considerable amount of time it took to deeply understand the original implementation.

Another modification done, aiming at reducing the execution time even more, was to develop a module written in C to deal with timestamps operations, because we noticed that these kinds of operations were commonly done by the EDBC rules.

The original ETALIS implementation is assigning a *datime/7* predicate to, for each event, internally represent the timestamp. The first six arguments are used to encode a date. For example, the date 1/Jan/1990:00:00:00 is represented by $datime(1990, 1, 1, 0, 0, 0, \_)$. The last argument is just a counter for the events with the same date.

This representation of a timestamp makes the code for the operations with dates, such as the addition of a certain amount of seconds to a timestamp, unnecessarily more complex, because it is necessary to take into account that the time units have a limit. For example, imagine that it is necessary to add 5 seconds to the following timestamp: $datime(1990, 12, 31, 23, 59, 59, \_)$. The procedure to make that operation would need to consider the cases when the time units exceed its limits, making them affect the time of the next time units.

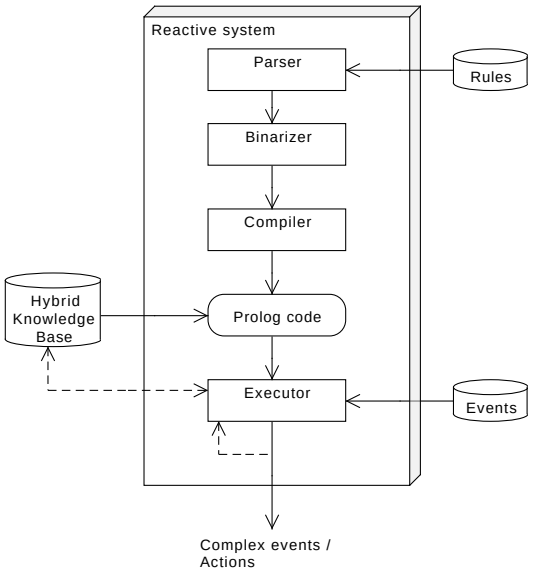
Instead of this format, we decided to use *epochs* to represent dates. An epoch, as used in Unix and POSIX-compliant systems, is the number of seconds elapsed since 1/Jan/1970:00:00:00.

Given this, the *datime/7* predicate was modified to *datime/2*, where the first argument is an *epoch*. The code for the operations with this predicate was reimplemented in C to provide full support for working with *epochs*.

After modifying the way the facts were stored, the next step was to start implementing the system. The main components that compose the original ETALIS system are the **parser**, the **binarizer**, **compiler** and the **executor**. Each of these components play an essential role in the way the system works. All of these components went through the "cleaning" phase, meaning that each component were left only with the parts that would be reused for our system. One can see in Figure 5.1 a simplified architecture of the reactive system.

To better understand what was developed in each of these components, the rest of this

Figure 5.1: Reactive system architecture



section is divided into several subsections, each dedicated to a component. But before going into detail about the system implementation, we start by presenting the final set of operators available in our system.

5.1.1 Rules and Operators

As mentioned before, the ETALIS system implements all thirteen relationships of Allen's algebra and some other conventional operators, such as the *and* or the *or* operators, which can be found in [AFRSSS10]. Since ETALIS is being used as the base of our CEP system, all operators used in ETALIS are maintained, because we believe that they cover a large range of interesting situations related with temporal reasoning, such as the occurrence of an event followed by another one or the occurrence of an event during the occurrence of another event. Besides that, those operators cover most of the events presented in the Snoop event algebra. There is only one operator that is present in Snoop but not in ETALIS. Since the Snoop event algebra is a very popular algebra in the literature, we have decided to include the missing operator that is not covered by the ETALIS system, namely the so-called *Periodic* operator. All these operators are described in Table 5.1 [AFRSSS10].

The complex events are defined using *patterns*, which in turn are constructed using this set of operators. The rules are very similar with the Prolog syntax, having the following format:

$$Event \leftarrow Pattern.$$

The left-hand side of the rule contains the complex event that is being defined. The right-hand side contains the *pattern* which is used to describe the situation in which the complex event should be triggered, utilizing the available operators.

For example, a rule to express that an event `a` is triggered when an event `b` is followed by an event `c` can be written as follows:

```
1    a <- b seq c.
```

The event `a` corresponds to the definition of the complex event, whereas `b seq c` corresponds to the *pattern*. In this particular case, the *pattern* is formed using a single operator, but the user can combine as many operators as needed to construct the *pattern*. Also, the events used in the right-hand side of the rules can be atomic events or other complex events already defined.

The events, atomic or complex, can carry additional information. This can be provenance data, numerical data or other kinds of information. For instance, an event representing a measurement from a temperature sensor can be represented as $tmp(n)$, where n is numerical data containing the measurement.

The operators presented here are the operators that our system offers to combine events in order to form new ones. Compared with the ETALIS system, our contribution is the addition of the *periodic* operator.

5.1.2 Parser and Binarizer

In this section, two of the main components are discussed, the **parser** and the **binarizer**. The **parser** is used to check if the rules are all written according to the syntax, while the **binarizer** aims to split the rules into several auxiliary rules, making each rule only containing patterns with one operator.

After the rules are defined by the user, they are sent to the first component of the CEP system, the **parser**. This component is responsible for checking if the rules are all written according to the defined syntax. If so, they are converted into an internal format, represented by the predicate `eventClause(Label,Head,Pattern)`. This is a meta-predicate that keeps, for each rule, what complex event (*Head*) is being defined and how it is defined (*Pattern*). The *Label* is just an identifier that the user can give to a rule. If any identifier is not assigned to a rule, then the system uses the predefined label *unlabeled*. So, after the *parsing* phase, this component sends to the *binarizer* a set of meta-predicates. Example 6 shows the final result of a *parsing* phase.

Example 6. The following user-defined rules:

```
1  a <- a0 seq a1.
2  b(X,Y) <- b0(X) and b1(Y).
3  c(X) <- c0(X) or c1(X) or c2(X).
```

are converted into the following internal rules:

```
1  eventClause(unlabeled, a, seq(a0,a1)).
2  eventClause(unlabeled, b(X,Y), and(b0(X),b1(Y))).
3  eventClause(unlabeled, c(X), or(c0(X),or(c1(X),c2(X)))).
```

Table 5.1: Operators supported by our CEP system.

Operator	Representation	Description
Sequence	`a SEQ b`	An event is triggered when the event `b` occurs after the event `a` had occurred.
Conjunction	`a AND b`	An event is triggered when both event `a` and `b` occurs, independently of the order of occurrence.
Concurrent conjunction	`a PAR b`	It has the same behavior of previous operator, but in this case the interval of occurrence of both events `a` and `b` have to intersect.
Disjunction	`a OR b`	An event is triggered when either `a` or `b` occurs.
Equals	`a EQUALS c`	An event is triggered when both `a` and `b` occur with the same initial and final time.
Meets	`a MEETS b`	An event is triggered when `b` has the initial time equal to the final time of `a`.
During	`a DURING b`	An event is triggered when the time of occurrence of `a` is contained in the time of occurrence of `b`.
Starts	`a STARTS b`	An event is triggered when `a` and `b` have the same initial time and `a` finishes before `b`
Finishes	`a Finishes b`	An event is triggered when `a` and `b` have the same final time and `a` starts after `b`.
Not	`a NOT b`	An event is triggered if the event `b` does not occur during the event `a`
Periodic	`Periodic(`t_i`,s,`t_f`)`	An event is triggered in every step `s` starting after time t_i and ending before time t_f.
Periodic	`Periodic(a,s,b)`	An event is triggered in every step `s` starting after the event `a` and ending before the event `b`.

Since our system supports an operator that is not originally supported by the ETALIS, the **parser** component had to be modified.

After the *parsing* phase, the produced rules are sent to the *binarizer*. This component initiates the conversion of the user-defined rules into EDBC rules. It starts with a process called *binarization* and aims at converting rules patterns into binary rules. As explained in Section 2.1.4, there are several advantages in using this method, such as the increase of the possibility of sharing rules among events, ease in rules management and ease in the implementation of binary operators, instead of operators with a variable number of events. For example, the following rule:

```
a <- b seq c seq d.
```

should be converted into:

```
a <- tmp seq d.
tmp <- a seq b.
```

But how does this process work? The *binarizer* component iterates over a set of parsed rules (the ones returned by the *parser*) and for each rule it checks the rule body if it needs to be decomposed into binary rules. If this is the case, then it executes a recursive procedure to decompose the rule body into binary rules. This procedure returns a set of rules that contains all the binary rules from which it is possible to obtain the original rule again.

Since having repeated rules is not efficient, in terms of space, the binary rules produced, for each original rule, are compared with the set of the already existing binary rules. This comparison aims to avoid storing binary rules that share the same body. To make this *binarization* process more clear, imagine the following set of rules:

```
a0 <- b seq c seq d.
a1 <- b seq c seq e.
```

One can see that both rules share a part of their bodies (`b seq c`). The first rule, after passing through the described process, yields the following rules:

```
a0 <- tmp1 seq d.
tmp1 <- b seq c.
```

When the second rules passes through the same process the following rules will be created:

```
a1 <- tmp2 seq e.
tmp2 <- b seq c.
```

However, we can see that the bodies of the rules with heads *tmp2* and *tmp1* coincide. Instead of storing the *tmp2* rule, the *binarization* process ignores that rule and uses the one previously defined (*tmp1*). The final result will be the following:

```
a0 <- tmp1 seq d.
a1 <- tmp1 seq e.
```

```
3 tmp1 <- b seq c.
```

When the process reaches the end, a set of binary rules are passed to the next component, the **compiler**.

But before going to the next component, it is important to point out that the implementation of this process did not have full support by the ETALIS team. The routine that checks whether some binary rule already occurs in the set of binary rules is only implemented for the YAP and SWI engines. For all the other engines, no such procedure was implemented. Since our system utilizes the XSB engine, we had to implement a predicate to compare rules bodies. This predicate can be seen as an extension of the `member/2` predicate available as a Prolog standard. Since we wanted to compare rules bodies, which are not grounded, we could not simply use the == operator. Instead, a *variant* test had to be used. This technique allows us to compare two predicates to check if they are the same. For example, these two predicates `p(A,B)` and `p(C,D)` are variants, while these two predicates `p(A,A)` and `p(B,C)` are not.

Example 7, which is the continuation of Example 6, shows using real code the resulting rules of *binarization* process.

Example 7. The following parsed rules:

```
1 eventClause(unlabeled, a, seq(a0,a1)).
2 eventClause(unlabeled, b(X,Y), and(b0(X),b1(Y))).
3 eventClause(unlabeled, c(X), or(c0(X),or(c1(X),c2(X)))).
```

are converted into:

```
1  %Rule: a <- a0 seq a1.
2  eventClause(unlabeled,a,seqf(a0,a1)).
3
4  %Rule: b(X,Y) <- b0(X) and b1(Y).
5  eventClause(unlabeled,temp_e_1(b0(_h295),b1(_h309)),
6      andf(b0(_h295),b1(_h309))).
7
8  eventClause(unlabeled,b(_h295,_h309),temp_e_1(b0(_h295),
9      b1(_h309))).
10
11 %Rule: c(X) <- c0(X) or c1(X) or c2(X).
12 eventClause(unlabeled,temp_e_2(c0(_h441),c1(_h441)),
13     orf(c0(_h441),c1(_h441))).
14
15 eventClause(unlabeled,
16     temp_e_3(temp_e_2(c0(_h441),c1(_h441)),c2(_h441)),
17     orf(temp_e_2(c0(_h441),c1(_h441)),c2(_h441))).
18
19 eventClause(unlabeled,c(_h441),
20     temp_e_3(temp_e_2(c0(_h441),c1(_h441)),c2(_h441))).
```

5.1.3 Compiler

Up to now some earlier components were presented aiming to convert the user-defined rules into an internal format and split them into binary rules. The first component to take action is the **parser** which goal is to check the rules syntax and transform them into an internal format. The **binarizer** is the next component and its goal is to transform all the rules into binary rules. The final component in the rules' processing phase is the **compiler** which is going to be described in this section.

The goal of this component is to transform the binary rules into EDBC rules. To accomplish that, several algorithms need to be defined in order to transform the rules. Each operator has a specific algorithm because the behaviour of each operator is different. However, since the ETALIS system is being used as the basis of our CEP system, the algorithms are already defined for each operator present in ETALIS. Those algorithms are not going to be described here, because they are all presented in [ARFS12].

Since we contributed with a new operator, that is going to be the only one whose algorithm will be detailed here. But before presenting the algorithm for the *periodic* operator, it is better to begin with an explanation of how the rules are represented internally, after the *compilation* phase.

To give a first intuition of how the user-defined rules are represented internally after being transformed into EDBC rules, consider the following example:

Example 8. Consider the binary rule `eventClause(unlabeled,a,seqf(a0,a1))` taken from Example 7. After applying the correspondent algorithm, the rule is transformed into the following set of EDBC rules:

```
1 trClause(unlabeled,event(a0,[T1,T2]),
2     ins(goal(event(a1),event(a0,[T1,T2]),event(a)))).
3
4 trClause(unlabeled,event(a1,[T3,T4]),
5     seqf(db(unlabel,goal(...)),del(unlabel,goal(...)),
6         less(T2,T3), event(a)).
```

The first rule corresponds to the event `a0` and can be interpreted as "when the event `a0` occurs, insert into the system the goal `goal(event(a1),event(a0,[T1,T2],event(a)))`", which can be read as "the event `a0` occurred within time T1 and T2 and we are waiting for the event `a1` to occur to trigger the event `a`". Thus we know that after the event `a0` occurs, the system will have in its internal state the predicate `goal/3`.

The second rule corresponds to the event `a1` and can be interpreted as "when event `a1` occurs, check if event `a0` has occurred, and if so remove it from the internal state, check if its final time is less than the initial time of `a1` and trigger the event `a`". Checking for the occurrence of the event `a0`, means that it will look for the *goal* predicates containing the event `a0`. After the event `a1` occurs the system will no longer have the information that `a0` occurred and the event `a` will be triggered.

Once again, this component has a similar behavior as the other components, in the

sense that it is also an iterative process. It receives a set of binary rules as input and for each rule the corresponding algorithm is applied according to the operator that is present in that rule. As shown earlier, the binary rules are represented using the meta-predicate `eventClause/3`. However, after applying the algorithm to the binary rules, two or more new rules are produced and they are encoded by the meta-predicate `trClause/3`. For example, the binary rule from Example 8 produces two new EDBC rules.

The meta-predicate for the EDBC rules has the following format: `trClause(Label, Event, Goals)`, where the *Label* is the same label present in the binary rules, *Event* is one of the events present in the rules' body and *Goals* are a sequence of goals to be executed when the event *Event* is triggered.

When the *compilation* process ends, each binary rule gives rise to a new set of rules. One may be wondering why one simple rule produces several rules. The answer is discussed in Section 2.1.4. When the EDBC rules were presented, we explained that this type of rules falls into two types. The first one is called the *goal inserting rule* and it is used for those events that start a *pattern*, i.e., the events that begin the detection of some complex event but are not sufficient to trigger it. The first EDBC rule from Example 8 is an example of a *goal inserting rule*.

The second type is called the *checking rule* and it is used when the event needed to complete the *pattern*, for some complex event, occurs. It checks if the first event occurred, triggers the complex event and deletes from the internal state all the facts that were "waiting" for this event. The second EDBC rule from Example 8 is an example of a *checking rule*.

These two kinds of rules, as said previously, are encoded in the meta-predicate `trClause/3`. But how does this meta-predicate differentiate the two kinds of rules? The main difference is in the argument *Goals*. For the first type of rules this argument will contain an insertion action, i.e. when the event occurs some goal will be asserted in the system. For the other kind of rules this argument will contain a sequence of actions to be performed. This set of actions depends on the semantics of the operators used to construct the complex event.

At this point, the reader has a notion of how the EDBC rules are encoded internally by the system. Thus, the conditions are met for explaining our algorithm to convert the rules using the *periodic* operator into a set of EDBC rules.

Our periodic operator is a special kind of operator, in the sense that it is not used to define a complex event combining other events, but instead it is used to trigger an event periodically during a time interval. This interval can be of two types: an interval defined by an initial and a final timestamp or an interval defined by two events. The most trivial case to implement is when the time interval is defined by two specific timestamps, because it is known a priori when the periodic event should start and when it should stop. When the time interval is defined by two events there is no way to know when those events will occur and hence it makes the implementation more complex for that

particular case.

Starting from the most simple case, the transformation can be seen in Algorithm 1. For the sake of readability, the algorithm is shown in pseudo-code.

Algorithm 1 Periodic

Input: event rule `a <- periodic(date1, s, date2).`
Output: EDBC rule for periodic operator.
Each event `a <- periodic(date1, s, date2)` is converted into: {

$$a(T_1, T_2) : -T_3 = T_2 + s,$$
$$T_3 < date2,$$
$$reg\_periodic(event(a), T_2, s).$$

}

This algorithm will produce one single `trClause/3` meta-predicate for the event that is being triggered. What this algorithm means is that when the event `a` occurs, the next occurrence time (T_3) is computed and it is checked if that time is smaller than the time that defines the end of the time interval for the periodic event, so the event `a` is registered to be triggered at time T_3. This algorithm might sound a little bit odd, because when the periodic event occurs it registers itself to occur on the next step.

It might seem that there is an infinite loop, but the condition $T_3 < date2$ is what prevents the existence of the endless loop. Yet there is another particularity in this algorithm. In the algorithms for the other operators, each EBDC rule is "waiting" for some event to occur. However, for this operator, the only EDBC rule created is "waiting" for the event that we want to be triggered periodically. Now the question is how is the event triggered for the first time? During the *compilation* process, and even before the EDBC rule is created, the system records the event `a` to be triggered for the first time at the correct time.

The transformation for the more complex case of the periodic event can be seen in Algorithm 2.

Algorithm 2 Periodic

Input: event rule `a <- periodic(e1, s, e2).`
Output: EDBC rule for periodic operator.
Each event `a <- periodic(e1, s, e2)` is converted into: {

$$e1(T_1, T_2) : -reg\_periodic(event(a), T_1, s).$$
$$a(T_3, T_4) : -reg\_periodic(event(a), T_3, s).$$
$$e2(T_5, T_6) : -unreg\_periodic(event(a)).$$

}

In this case there is no specific time interval for the periodic event. The periodic event only starts after some event is triggered and ends before another event is triggered. Thus, for this type of periodic event, three `trClause/3` meta-predicates need to be created. The algorithm tells us that when the event `e1` is triggered the event `a` is registered to be triggered at the time $T_1 + s$. After that, the event `a` is periodically triggered every s

steps. The only way to stop the periodic event is that event `e2` occurs. It indicates that from the time defined by T_5 the event `a` will no longer be triggered periodically. Despite the algorithm seeming simple, the procedures for registering and unregistering an event are not trivial. Since it is not known when the event `e2` will occur, a stop condition for the periodic event cannot be established a priori for the periodic event. Instead the stop condition is dependent on the occurrence of the event `e2`.

After the *compilation* process is complete, the rules do not need to pass through another modification process, instead they are all stored internally by the system.

In both algorithms for the *periodic* operator, registering and unregistering events is a necessary task to be performed. But what is the meaning of registering or unregistering an event? The process of registering an event is done using the predicate *reg_periodic/3*. This predicate is used for adding the periodic event into a queue, waiting for the time to be triggered. The process of unregistering an event is done using the *unreg_periodic/3* and it is used for the opposite purpose, i.e., to remove the periodic event from the queue.

Both predicates were developed by us and they are explained in more detail in the next section, since they are used during the *execution* phase. But before, it is important to point out that the process of understanding and extending the **compiler** module was not a trivial task. The algorithms for the *periodic* operator might seem straightforward but the process of understanding the way the EDBC rules were implemented was a difficult task. Mostly because there is no documentation explaining how the pseudo-code, for each algorithm presented in the ETALIS papers, is implemented in practice. We felt that there is a considerable gap between the theory of ETALIS and its implementation.

While testing if the operators were all correctly implemented, we found that the algorithm for the *equals* operator was not correctly implemented.

According to the definition of the *equals* operator, two events are equal if they happen at the same time interval. However the ETALIS implementation for this operator was not following the definition correctly, making their implementation dependent on the order the events participating in the equality are processed. Consider the following set of rules:

```
1 a <- a0 seq a1.
2 b <- a0 seq a1.
3
4 c <- a equals b.
```

In this scenario, both events *a* and *b*, when triggered, will have the same occurrence interval. If the event *a* is the first one to be processed by the system, the event *c* would be triggered. However, if the event *b* is the first one to be processed, the event *c* would not be triggered, when it should be. For that to happen, one would need to redefine the rule for the event *c*, changing the order the events *a* and *b* appear in the operator *equals*.

To rectify this bug, we extended the algorithm for the *equals* operator to also cover the case when the event *b* is processed in the first place.

5.1.4 Executor

In the previous section it was explained how the EDBC rules are represented internally and what was the proposed solution to transform the periodic event rules into EDBC rules.

The **compiler** component is the last component that modifies the rules and when it is done transforming the rules, it stores them to be used by the system. We are now at a point where the system enters the passive mode, i.e., after processing the rules, the **executor** component handles arriving events.

The **executor** is the last main component of those that composes the CEP system. Its goal is to process the events that arrive, find the EDBC rules that depend on them and (possibly) trigger some complex event. Throughout this section, we are going to explain how the *executor* finds the EDBC rules and what it does to trigger events.

When an event arrives, the system has to look up EDBC rules for that particular event. The result of this search for the EDBC rules has two possible outcomes, the case when the event is part of some complex event and the case that it is not part of any complex event.

The most trivial case to treat is when it does not belong to any complex event. In that case, the search for the EDBC rules will return an empty list (meaning that it does not have any EDBC rule) and the event is discarded.

The more interesting case is when the system receives an event that belongs to the definition of some complex event. In this situation, we know that the event will have at least one EDBC rule belonging to it, meaning that when the **controller** looks for the EDBC rules it will return at least one.

The result of this search is a set of `trClause/3` rule heads. As explained earlier, these meta-predicates are used by the system to encode the EDBC rules. Their third argument is used to place a set of goals to be executed by the system, when the event corresponding to that meta-predicate is triggered.

The set of goals can vary depending on the type of the EDBC rule and depending on the operator associated to the event corresponding to the rule. The goals can be that some fact is asserted into or retracted from the internal state of the system but it can also be a query to the internal state, trigger an event and so on.

The **executor** component is then responsible for executing all the goals from each EDBC rule found.

Summing up, the idea of the **executor** is quite straightforward. It receives an event, searches for EDBC rules and executes the actions of those rules.

Earlier in this chapter, it was discussed that the predicate *findall* is not efficient when using the XSB standard storage mechanism compared with the trie storage. For that reason, we changed the declaration of the facts to be stored using the trie storage mechanism.

The most significant change, related with the storage mechanism, was precisely done in this component. In the procedure to look for EDBC rules, we noticed that the original

ETALIS implementation was using the *findall* predicate to collect all the *trClause* meta-predicates, for each event that was arriving. These meta-predicates were declared to be stored using the standard mechanism.

The *trClause/3* facts represent an essential part of the CEP system, because they are used by the system to maintain the definitions of the complex events. With a high rate of arrival of events, the system is going to spend most of its time collecting these instances. So it is important that this search procedure can be as much optimized as possible.

Once again, we believe that the ETALIS team concern was to support as much Prolog engines they could, or at least the most popular ones. This choice has consequences and one of them is precisely not being able to explore the features that each engine offers. Since our system is being entirely developed for XSB, we could take advantage of the trie storage mechanism to improve the storage of the *trClause* predicates, making it being stored using tries.

5.1.5 Periodic Operator

Having explained how the **executor** component works, we are now able to analyze in more detail what is our approach to support the *periodic* operator and how that operator works during the system execution.

In order to be able to receive external events and at the same time having an event being triggered periodically it is necessary to use more than one single thread, because having a periodic task, such as a periodic event being triggered, requires a process with an active approach, i.e., that is constantly checking for the right time to trigger the periodic event. That active approach is the opposite approach required to wait for the arrival of events.

To combine both approaches in our system, we came up with a solution to introduce several concurrent processes to deal with the arrival of external events and the periodic events. One of the concurrent processes would be responsible for waiting for the events' arrival. A second one would be constantly checking for the expected time to trigger the periodic events and a third one would be responsible for processing the events. This solution is possible because the XSB engine offers a mechanism for *multi-threading* programming, giving us high-level primitives to work with POSIX threads. The scheme of our solution is depicted in Figure 5.2. This scheme shows how the **executor** component is structured when a periodic event exists.

The *External Thread* is the process that waits for the external events to arrive, and when they do, it sends them to the main process, the *Events Thread*. The *Periodic Thread* is the process responsible for triggering the periodic events. This is the process that takes an active approach, i.e., it is constantly checking if there is periodic events to trigger. When it finds one, it also sends it to the *Events Thread*. This *Events Thread* is like the normal **executor** component, when there is no periodic events. It receives both external events and periodic events and it looks for EDBC rules for those events. Since the periodic

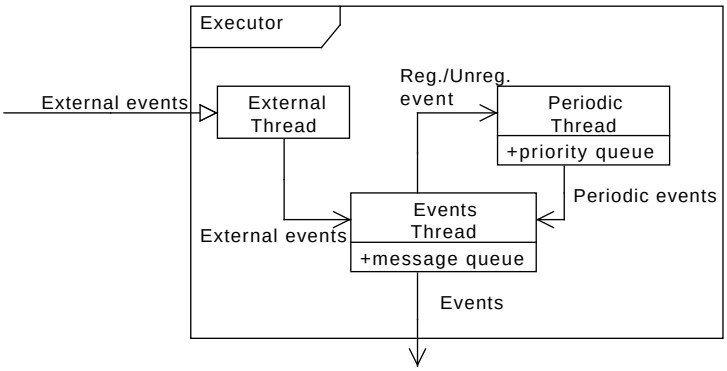


Figure 5.2: Executor component architecture for periodic events

events needs to be registered again, after they are triggered, this component also tells the *Periodic Thread* to register a specific event. The same way it is responsible for keeping the periodic events "alive", it can also make them stop from being triggered.

The *External Thread* is the most simple process of all three. It receives the external event and sends it directly to the *Events Thread*. It might appear to be a useless process, since its job is just to send events to another process. Indeed one may ask why the *Events Thread* cannot receive those external events and thus reduce the number of concurrent threads to two?

This cannot happen because of the way how the *Events Thread* works. In order to receive the two kinds of events, a message queue is being used. This type of queues is commonly used in the concurrent programming environment, when there is the need to exchange messages between threads. These queues have the advantage that when there is no messages in the queue, the *consumers* of the queue suspend their work and wait for the arrival of new messages. When a new message arrives, they immediately return to their work.

Taking into account the behaviour of these queues, it is impossible for the *Events Thread* to receive the external events directly, because it will not respond to any command unless there is a new message in the queue. Since an external source (the one that sends the external events to the CEP system) cannot send a message to the *Events Thread* message queue, it is necessary to create the *External Thread* to bridge the gap between the external source and the *Events Thread*. The pseudo-code for the *External Thread* is shown in Algorithm 3.

Algorithm 3 External Thread pseudo-code

```
procedure EXTERNALEVENT(Event)
    thread_send_message(events, Event)
end procedure
```

For the *Periodic Thread*, we decided to implement a simple priority queue to maintain

the periodic events that are waiting to be triggered. Each entry in the queue is ordered by the time it should be triggered, so we know that at the head of the queue is always the next event to be triggered. Then, the *Periodic Thread* continuously checks if it is already time to trigger the event in the head of the queue, if so, it sends it to the *Events Thread* process, also using the message queue. The pseudo-code is shown in Algorithm 4.

Algorithm 4 Periodic Thread pseudo-code

```
procedure CHECKPERIODICEVENTS
    while true do
        with_mutex(pq_mutex, get_head_pq(Head))
        if Head ← ∅ then
            thread_sleep(1000)
            CheckPeriodicEvents()
        else if Head ← event(Event, Next_TS) then
            sleep ← current_TS − Next_TS
            if sleep > 0 then
                thread_sleep(1000)
                CheckPeriodicEvents()
            else
                with_mutex(pq_queue, pop_pq(event(Event, Next_TS)))
                thread_send_message(events, event(Event, Next_TS))
            end if
        end if
    end while
end procedure
```

Our system, by default, does not maintain the configuration depicted in Figure 5.2, having only one process. So how does it change to this configuration when there are periodic events to be triggered?

It is during the *compilation* phase that the system realizes that there will be periodic events. When the **compiler** finds the rules for the periodic events then, even before applying the algorithm to those rules, it activates a flag that indicates that there will be periodic events and signals the **executor** to change its structure. When the **executor** receives that signal, it changes its initial structure with one single thread to the one shown in Figure 5.2. The **executor** will keep that structure until the whole system terminates, even if periodic events occur during a small period of time.

Throughout this section and the previous one, we have been talking about the process of registering and unregistering periodic events. It was explained that this process is used by the system to register the events to be triggered in the correct time or to unregistering them, making them no longer to be triggered.

The pseudo-code for the registering procedure is quite straightforward and can be seen in Algorithm 5.

The procedure receives as input the event to be triggered ($Event$), the last timestamp

Algorithm 5 Periodic event registering procedure

procedure REG_PERIODIC($Event$, Ts , S)
$Next\_ts \leftarrow Ts + S$
$with\_mutex(pq\_mutex, push\_pq(Event, Next\_ts))$
end procedure

the event was triggered (Ts) and the temporal step (S) of the periodic event. The procedure starts by calculating the next timestamp the event should be triggered and then inserts the event into the priority queue.

The predicate *with_mutex/2* is one of the primitives that XSB offers to be used in the *multi-threading programming* environment. It is used to guarantee exclusive access to the priority queue, since there are two processes accessing the priority queue (the *Events Thread* and the *Periodic Thread*).

The unregistering procedure is not used by the two types of periodic events. When the periodic event is defined by a time interval, it is possible to predict the time the periodic event has to stop. However, for the other type of periodic events there is no way to know, a priori, when it should stop, meaning that the periodic event is continuously being registered to be triggered until the event that dictates the end of the periodic event arrives.

The procedure *unreg_periodic* is then used, by the event that dictates the end of the periodic event, to prevent the periodic event from being triggered. The pseudo-code for this procedure can be seen in Algorithm 6.

Algorithm 6 Periodic event unregistering procedure

procedure UNREG_PERIODIC($Event$)
$removePeriodicFromPriorityQueue(Event)$
$removePeriodicFromMessageQueue(Event)$
end procedure

When the terminal event for a periodic event arrives, it is necessary to stop all the instances of the periodic event from being triggered. The methods *removePeriodicFromPriorityQueue(Event)* and *removePeriodicFromMessageQueue(Event)* are responsible for removing all the instances of the periodic events.

The process of structuring the architecture for the periodic event was not always a trivial task, because of the periodic events that are defined by two events. To better understand the difficulties that this type of event brought during the design of the architecture, consider the following case, where there is only a periodic event:

```
1   a <- periodic(start, 4, fini).
```

The event *a* is going to be triggered every 4 seconds after the event *start* occurs and before the *fini* event occurs.

Now consider this first scenario, depicted in Figure 5.3.

Figure 5.3: Periodic scenario 1

The *start* event is triggered at time 0 and from that time on, the *a* event is triggered every 4 seconds. When the *fini* event occurs at time 13, the *a* event is no longer triggered. In this scenario everything should go well. The problems arise when the events overlap.

This time, consider a difference scenario, depicted in Figure 5.4, where the event *fini* occurs at time 12.

Figure 5.4: Periodic scenario 2

As one can see, now the event *a* and *fini* occur at the same time, meaning that two processes are going to be executed at the same time. The *External Thread* is going to send to the *Events Thread* the event *fini*, while the *Periodic Thread* is going to try to send the event *a* to the *Events Thread*. Since both events happen at the same time, there is no way to know which event is going to be processed first.

According to our definition of the *periodic* event in Table 5.1, the *fini* event should be processed first in order to stop the event *a* from being triggered.

Events Thread implementation takes into account the solution for dealing with concurrency in this scenario. Algorithm 7 shows the pseudo-code for this thread.

The algorithm is composed of two main procedures: *GetMessage* and *ProcessEvent*. The first procedure is the first one to be called when the *Events Thread* begins its execution. It tries to get a message from the message queue using the *thread_get_message/2*. The message that the process is looking for is a message containing an external event or a periodic event, both contained in predicate *event/2*. If it finds any event in the message queue, it calls the *ProcessEvent* procedure to process the event. If if fails retrieving an event from the message queue, the *Events Thread* suspends its execution until an event arrives to be processed. Recall that, whenever a thread tries to get a message from this type of queues and there is no message in the queue, the thread automatically suspends its execution until a new message arrives.

The *ProcessEvent* procedure is then used to process an event. There are two cases to consider when processing an event, the case when the event is a periodic event and when it is not a periodic event.

The first case is when the event being processed is a periodic event. In this case, there

Algorithm 7 Events Thread pseudo-code

```
procedure GETMESSAGE
    thread_get_message(events, event(Event, TS))
    processEvent(Event, TS)
end procedure
procedure PROCESSEVENT(Event,TS)
    if isPeriodicEvent(Event) then
        Terminal ← getTerminalEvent(Event)
        if thread_peek_message(events, event(Terminal, TS)) then
            trigger_event(Terminal, TS)
            GetMessage()
        else
            trigger_event(Event, TS)
            GetMessage()
        end if
    end if
    trigger_event(Event, TS)
    GetMessage()
end procedure
```

are two possible situations. The first one is when there is a terminal event in the queue with the same timestamp, meaning that they both occurred at the same time. In that case, the terminal event is triggered instead of the periodic event, thus ensuring that the periodic event is no longer triggered. The second one is when the periodic event has not been stopped yet, meaning that it can be triggered.

The second case is when the arrived event is not a periodic event. In this case the event is normally triggered and then the *GetMessage()* procedure is called to get another event from the queue. Recall that, according to our Algorithm 2 for the periodic operator, when a terminal event is triggered, it will remove from the priority queue and from the message queue all the instances of the periodic event that is being stopped.

There is also a third situation that we had to consider when developing the support for the periodic event. Consider the scenario, depicted in Figure 5.5, where a second event *start* arrives.

Figure 5.5: Periodic scenario 3

Facing this scenario, when another initial event occurs, we decided to let the *start* event create another instance of *a*. Then, when the *fini* event arrives, all the instances of *a* will be stopped. Another approaches could be followed. For instance, we could let the second *start* event stop the current periodic event and trigger a new one starting from the

time the second *start* event occurred. However, we think that our solution is the one that seems to make more sense, because it does not deviate from the original purpose of an initiator event, i.e., an event that should only start a periodic event, independently of the current periodic events.

During the implementation of the solution for the periodic operator, we noticed that XSB had a bug in the implementation of *thread_sleep/1*. The argument for this predicate should be a numerical value that represents the amount of time a thread should sleep. According to the documentation, the time has to be indicated in milliseconds, but when we tried to put the *Events Thread* sleeping for 500ms, XSB launched an error.

After making some tests, we noticed that XSB did not allow us to define sleeping times with a granularity in the order of milliseconds. For example, if we tried to put 1500ms it would raise the same error. If we put 1000ms instead it would not raise the error.

We looked into the XSB source code to find the bug and fixed it. The bug was related with a possible overflow of a variable of the type timespec structure. The bug was reported to the XSB developers along with the fixing.

5.2 ECA rules processing

The previous section was dedicated to explain in detail how the CEP system works. It is composed of several components that work collaboratively to detect the occurrence of complex events. The CEP system discussed is based on the ETALIS system implementation.

However our goal with this thesis is also to create a system that is not only able to detect complex events but also capable of reacting in the presence of events.

To meet that goal, we present here in this section our solution for the reactive part of our system. This part is no longer based on the ETALIS system, instead it was created from scratch.

As explained previously, an ECA rule has the following representation:

On event **If** condition **Then** action

and can be read as "on the occurrence of the *event*, if the *condition* holds, then execute the *action*". Due to the declarative nature of the logic paradigm, this behaviour can be easily expressed. Therefore, our first concern was to develop a proper representation for the ECA rules.

The chosen representation is the following:

```
on event --> if cond ; then action
```

As one can see this representation is very close to natural language. The *event*, which can be either a complex or an atomic event, is used to place the event that will trigger

Table 5.2: Syntax for each action

Action	Syntax	Description
Insertion	insert(x)	Inserts a fact or a rule x into the knowledge base
Retraction	remove(x)	Removes a fact or a rule x from the knowledge base

an action. The *cond* part is where one can place the condition that must hold in order to perform the action. Lastly, the *action* part indicates the action to be performed.

In our system, the actions cannot be arbitrary. As mentioned in previous sections, the actions supported by our system are the insertion and the removal of facts or rules into/from the knowledge base. Table 5.2 shows the syntax for each available action.

To better illustrate how an ECA rule can be written, Example 9 is used to encode an ECA rule using our system.

Example 9. Whenever a person has a blood pressure higher than 180 mm Hg, then that person is in a critical state.

```
1  on hasBloodPressure(X,Y) --> if Y>180 ; then insert(criticalState(X)).
```

Having presented the syntax for the ECA rules, we are now able to explain in detail how we developed this reactive part of our system.

First, notice that these kind of rules have a behaviour that follows the event-driven computation approach. This behaviour is also present in the complex event rules. Since the logical paradigm is also being used to implement the reactive part, our approach was to also use the EDBC rules to encode the ECA rules.

To encode the ECA rules into EDBC rules the algorithm had to be implemented to make that translation. The algorithm, whose pseudo-code is shown in Algorithm 8, is similar to the ones developed for each operator in the CEP system.

Algorithm 8 Event-Condition-Action transformation

Input: ECA rule `on a -> if cond ; then action`
Output: EDBC rule for ECA rule.
Each event `on a -> if cond ; then action` is converted into: {
$a(T_1, T_2) :- check(cond),\ execute(action)$
}

To give support to the reactive rules, our CEP system had to be extended in the following modules: **parser**, **compiler** and **executor**. The **parser** module had to be extended with the new primitives to declare the ECA rules, such as the keywords *on*, *if* and *then*. The **compiler** component is where we had to add the algorithm to transform the ECA rules into EDBC rules. Finally, the **executor** module had to be extended to also look for the EDBC rules related with the ECA rules, when an event arrives.

The modification on the **executor** module raises an important point for discussion. As explained, the EDBC rules for the complex event rules were encoded internally using the meta-predicate *trClause*. However, for the ECA rules, the same meta-predicate could

not be used to encode the rules internally. Instead, the *trClauseECA* was chosen to encode the ECA rules.

When the **executor** receives an event, it will now look for the stored *trClause* and *trClauseECA* meta-predicates, returning two different sets of meta-predicates. The reason to separate the two kind of rules, through the use of different meta-predicates, is due to the *conflict resolution strategy*.

Conflict resolution is the process of selecting one or more rules from the *conflict set* to be fired. The *conflict set*, in our system, is a set composed of two subsets of rules, one containing the *trClause* meta-predicates and another containing the *trClauseECA* meta-predicates.

Example 10. Consider the following set of EDBC rules stored internally in our system:

```
1 trClause(Label,event(a,[T1,T2]), goal(...)).
2 trClauseECA(Label,event(a,[T1,T2]), goal(...)).
```

The event *a* is present in the definition of a complex event and is also used in an ECA rule, resulting in two EDBC rules.

When the event *a* arrives, the **executor** will look for the EDBC rules for that event and will return two different sets. One containing the set of *trClause* meta-predicates and another one containing the set of *trClauseECA* meta-predicates. Now, which of the sets should be processed first?

The *conflict resolution strategy* adopted in our system gives priority to the ECA rules over the complex event rules, meaning that the set with the *trClauseECA* will be processed first.

Example 11. In the following example, a case is shown where the order of the execution of both kinds of rules is important.

```
1 c <- b seq a.
2 on c --> if a ; then insert(c).
3 on a --> if not a ; then insert(a).
```

When the event *a* arrives, assuming that event *b* already occurred, the complex event *c* will be triggered which in turn triggers the ECA rule in (2) while the one in (3) is triggered by *a*. Assuming our *conflict strategy*, the **executor** will opt to execute the ECA rule for the event *a* before triggering the complex event *c*, meaning that the fact *a* will be inserted into the knowledge base. Only then the complex event *c* will be triggered along with the corresponding ECA rule, resulting in the insertion of the fact *c*.

However, if we would adopt the opposite *conflict strategy*, the complex event *c* would be triggered first and, as a consequence, the ECA rule for the complex event *c* would be also triggered. Only after the ECA rule for the event *a* would be triggered. In this case, the only fact inserted into the knowledge base would be *a*, while with our strategy both facts *a* and *c* would be inserted.

The strategy adopted in our system is not the strategy that all reactive systems should follow. In fact, none of the strategies discussed in Example 11 is the more correct. The right strategy depends on the context in which the reactive system is being used. Despite adopting only one strategy, our system can be easily extended with new strategies in future versions.

It is also important to point out that, similarly to the meta-predicate *trClause*, the *trClauseECA* meta-predicate is also declared to be stored using the trie storage mechanism.

5.3 Conclusion

Throughout this chapter, the development of a reactive system for the XSB Prolog engine was described. We started by discussing the implementation of the CEP system, which was implemented having as its base the ETALIS system. The implementation was not a trivial task. The lack of documentation made us spend a considerable amount of time understanding how the system was implemented.

However, that time spent studying ETALIS gave us a valuable insight about the way the system was implemented. The acquired knowledge from that studying phase, became important when implementing the reactive part of our system, allowing us to speed up the implementation of that part.

Also, during the implementation of our system, we always tried to take advantage of the features available in XSB, instead of relying on the standard mechanisms shared amongst the Prolog engines. In particular, the trie storage mechanism was used in an attempting to speed up the process of looking up for the EDBC rules.

The *multi-threading programming* mechanisms became also valuable to implement the *periodic* operator. Without the use of these mechanisms, it would be much harder to implement such an operator.

6

Protégé Plug-in

The previous chapter was dedicated to explain how the reactive system was implemented and how it works internally during its execution. The system is capable of combining atomic and complex events to form new events. Those events can then be used to trigger actions over the knowledge base.

However, our goal is to create a reactive system capable of working in the Semantic Web environment, in particular capable of combining ontologies with non-monotonic rules, forming a hybrid knowledge base, and react over it.

In this chapter will be explained our solution for the reactive hybrid knowledge base. In particular, will be explained how the Protégé plug-in was developed and how it works, using the reactive system described in the last chapter.

Since the plug-in is developed using Java, it is necessary to use a tool to make possible the communication between the plug-in and the reactive system. For that end, InterProlog, which is an open source Java front-end and functional enhancement for standard Prolog engines, such as XSB Prolog, was adopted.

In Figure 6.1 is depicted the Protégé plug-in architecture, whose main components and their dependencies are represented. The **GUI** component is responsible for creating the graphical interfaces to provide the communication between the user and the Protégé plug-in. The **Controller** component is the one responsible for controlling all the information flows between the components. The **Translator** component is used to translate the ontology and non-monotonic rules as well as the actions to be performed over the hybrid knowledge base. Lastly, the **Reactive System Interface** component is used to communicate with the **reactive system**.

As we did previously, this chapter is structured into several sections, each detailing a part of the development process of the Protégé plug-in.

Figure 6.1: Protégé plug-in architecture

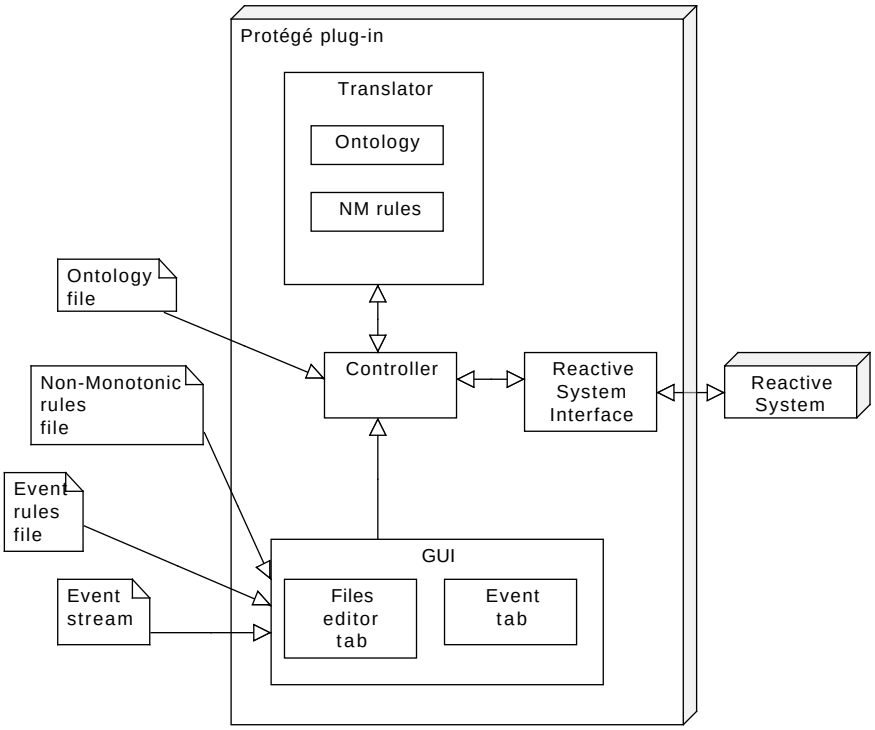


Figure 6.2: Protégé plug-in tabs

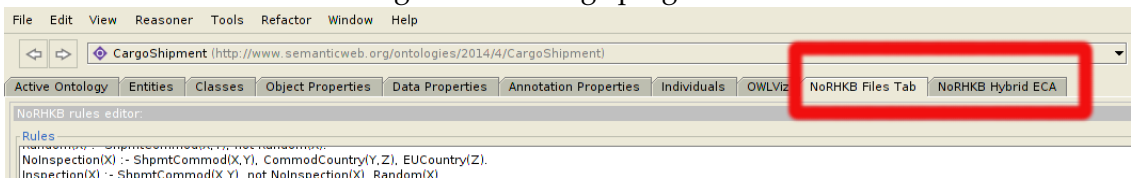
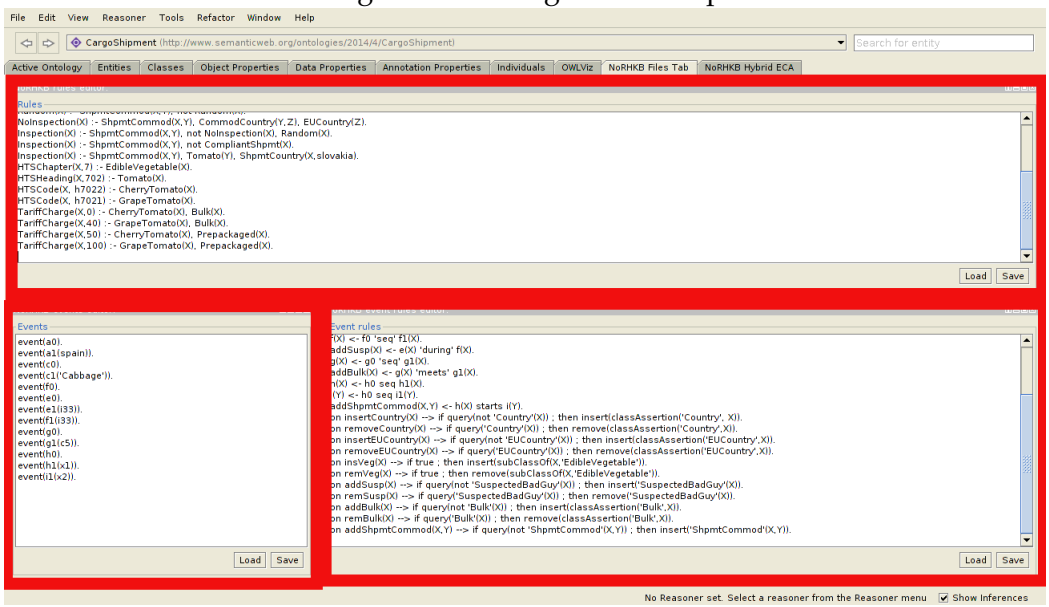


Figure 6.3: Protégé view components



6.1 GUI

Our initial task, in the plug-in development process, was the graphical interface design. Protégé can be extended with new plug-ins aiming to enhance its functionalities. These plug-ins come with new interfaces designed to give the users an easy way to explore the features presented in those plug-ins.

When developing a new plug-in, it is necessary to choose the type of it. There are two types of plug-ins that can be developed: *Core plug-ins* and *OWL plug-ins*. The *Core plug-ins* are the ones that add new features to Protégé, while the *OWL plug-ins* are the ones that add new options when creating/editing ontologies. Since we want to add a new feature to Protégé, our plug-in belongs to the first type, the *Core plug-ins*.

Inside the *Core plug-in* several components can be used to create an interface for the new plug-in. For our purpose, we chose to add two *WorkspaceTab* components. These components aim to make two new workspace tabs available in Protégé (Figure 6.2).

To make these two new tabs appear in Protégé it was not necessary to write any code, instead we just had to create a file called *plugin.xml* where the following had to be placed:

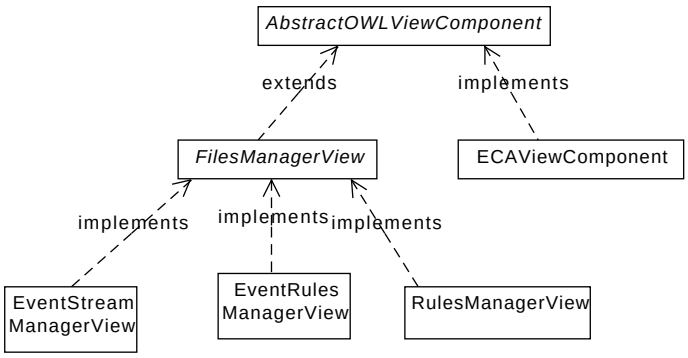
```
1    <extension id="FilesManagerTab"
2        point="org.protege.editor.core.application.WorkspaceTab">
3    <label value="NoRHKB␣Files␣Tab"/>
4    <class value="org.protege.editor.owl.ui.OWLWorkspaceViewsTab"/>
5    <index value="X"/>
6    <editorKitId value="OWLEditorKit"/>
7    <defaultViewConfigFileName value="viewconfig-filestab.xml"/>
8    </extension>
9
10   <extension id="HybridECATab"
11       point="org.protege.editor.core.application.WorkspaceTab">
12   <label value="NoRHKB␣Hybrid␣ECA"/>
13   <class value="org.protege.editor.owl.ui.OWLWorkspaceViewsTab"/>
14   <index value="X"/>
15   <editorKitId value="OWLEditorKit"/>
16   <defaultViewConfigFileName value="viewconfig-hybrideca.xml"/>
17   </extension>
```

The *plugin.xml* file contains a declaration of the ways in which the plug-in will extend the Protégé capabilities and (more advanced) the ways in which the plug-in capabilities can be extended by other plug-ins. These declarations indicates to Protégé to create two new empty tabs.

To fill these tabs with something that the user can interact with, it was necessary to add *ViewComponents* to the tabs. These components are the building blocks of workspace tabs. Each tab can contain one or more *ViewComponents*. Figure 6.3 shows the three components inside one of our tabs, the **Files tab editor**.

For our Protégé plug-in, four new views were created, where three of them are used to deal with the files, and one of them to interact with the reactive system.

Figure 6.4: Class diagram for the files components



Contrary to what was done for creating the two tabs, to create a new view is necessary to extend the *AbstractOWLViewComponent* class and implement the methods **initialiseOWLView** and **disposeOWLView**. The first method is called at the start of the plug-in instance life cycle, while the second one is called at the end of a plug-in life cycle, when the plug-in needs to be removed from the system.

As one can see in Figure 6.3, the three components responsible for managing the files have the same layout. To avoid having repeated code, we decided to create a general class to construct the views for managing the files. Thus, whenever a new component to manage files needs to be created, one can simply implement that general class without worrying about creating the view elements.

For the other component, the one responsible for interacting with the *reactive system*, we implemented directly the *AbstractOWLViewComponent* class. The class diagram for the view components can be seen in figure 6.4.

In the end, the resulting tabs for the plug-in, the **Files editor tab** and the **Event tab**, can be seen in Figures 3.3 and 3.4.

The **Files editor tab** is where one can load the event rules file, the non-monotonic rules and (possibly) the event stream file. None of the files are mandatory, however in most real situations the user probably wants to load at least the first two files. The third file, the event stream file, can be used as a place to put a set of events to be executed sequentially, instead of sending them manually one by one to the reactive system. The ontology is the only thing that is not loaded directly in this tab, because once we are working within a framework specifically for dealing with ontologies, the ontology is loaded using the existing Protégé tabs. The ontology and the files can be edited and saved before they are sent to the reactive system. This tab has no other use besides editing the files.

The **Event tab** is more interesting, because this is the place where one can interact with the reactive system. Once the user gives the instruction to start the system, the loaded files are sent to the **controller** component along with the ontology. After the system is completely initiated, it can start receiving events and at each event arrival, the result for that event is shown by the system. The result can be a detection of a complex event, a

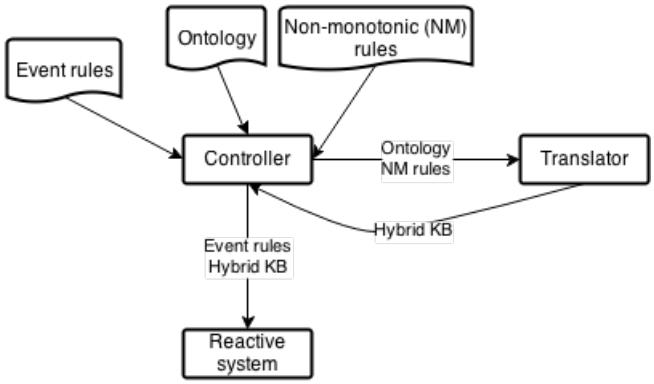


Figure 6.5: Controller initial phase

modification to the hybrid knowledge base or simply the indication that the event was triggered.

6.2 Controller

This is what is called the “brain" of the plug-in, i.e., this is the component responsible for controlling all the information flows between the other components.

Once the system has started, this component sends to the **translator** component the ontology and the non-monotonic rules to be translated. After the translation, all the translated rules are then sent to the reactive system. Both ECA rules and complex event rules do not need to be translated, because they are not part of the hybrid knowledge base, and thus they are sent directly to the reactive system to be processed. This initial situation is depicted in Figure 6.5.

When the initial phase is complete and the system is fully operational, the **controller** can start receiving events and sending them to the *reactive system*. As a consequence of an event arrival, an action can occur meaning that a modification is going to take place.

The *reactive system*, on the Prolog side, cannot perform the action directly over the knowledge base, because the hybrid knowledge base passed through a translation process before it was sent to the *reactive system*, meaning that any action to be performed over it has also to pass through the same process.

Every time an action has to be performed, the *reactive system* sends it to the **controller** component. After receiving the action, this component verifies if the actions is an axiom, meaning that it belongs to the ontology, or a fact/rule, meaning that it belongs to the set of non-monotonic rules. This distinction is necessary because the translation process is different for the cases when the action belongs to the ontology or to the set of non-monotonic rules. The actions’ translation process is explained in the next section. However, how does this component distinguish between an axiom and a rule/fact?

To be able to make that distinction, we developed a parser that recognizes the OWL functional syntax, so every time an action contains an axiom written using the OWL

functional syntax, it knows that that action is an axiom, otherwise it considers that the action is a rule or a fact. Despite this parser only recognizing these two kinds of actions, it can be easily extended to support actions to be performed to some other formalism, such as databases. Example 12 shows how one can perform actions to the two different formalisms.

Example 12. Consider the following two rules:

```
1 on hasBloodPressure(X,Y) --> if Y>180 ; then insert(criticalState(X)).
2 on hasBloodPressure(X,Y) --> if Y>180 ; then
3     insert(classAssertion('CriticalState', X)).
```

These two rules, besides having the same purpose, will have different practical results. The first one is used when one wants to insert a fact into the hybrid knowledge base, while the second one is used when one wants to insert an axiom into the hybrid knowledge base.

As one can notice, the second rule has a special predicate, the *classAssertion/2* predicate. This predicate is used to represent a class assertion, which is one kind of axioms. Whenever the **controller** component finds such predicates, it will know that the action is applied to an axiom. The axioms supported by our Protégé plug-in are listed in Appendix C.

We could have come up with a different solution to distinguish between the actions to be performed, however using the OWL functional syntax for the axioms seemed to be more intuitive to the one that writes the ECA rules and also easier to develop a parser for that type of syntax.

After parsing the action, if it is applied to an axiom, the parser returns an *OWLAxiom*, which is the basic unit of an ontology. Otherwise, if the action is applied to rule or a fact, the parser returns a *TermModel*, which is the basic unit used to represent facts and rules.

The resulting parsed action is then translated and sent to the *reactive system* to be performed over the hybrid knowledge base. However, before an action is performed, according to the ECA rules semantics a condition must hold, meaning that, in order to query the hybrid knowledge base, the query must also be translated to be in accordance with the syntax used for the hybrid knowledge base.

However, there are situations where there is no need to translate the queries. To distinguish both situations, we introduced a new meta-predicate (*query/1*) that can be used to refer to facts stored in the hybrid knowledge base. This meta-predicate indicates the *reactive system* that the fact contained in it needs to be translated, to be in accordance with the hybrid knowledge base syntax. To illustrate the need for this meta-predicate consider the Example 13.

Example 13. The following two rules show the need for a predicate to refer to facts stored in the hybrid knowledge base:

```
1 on hasBloodPressure(X,Y) --> if Y>180 ; then insert(criticalState(X)).
```

```
2 on remCriticalState(X) --> if query(criticalState(X)) ; then
3     remove(criticalState(X)).
```

In the first rule, the condition part of the ECA rule is not using any predicate present in the hybrid knowledge base, it is just an arithmetic comparison and thus it is not necessary to be translated. However, the second rule is referring to a fact that is stored in the hybrid knowledge base and thus it has to be translated first before it is posed to the background knowledge. While processing this ECA rule, when the *reactive system* finds the *query/1* meta-predicate, it knows that the content of this meta-predicate has to be sent to the **translator** before being posed against the hybrid knowledge base.

6.3 Translator

The main purpose for this component is to translate both ontology and non-monotonic rules into a set of rules compatible with XSB Prolog syntax. This component was partially developed by the NoHR team, however it was designed for a system whose goal is to query the hybrid knowledge base. This means that the translator was not prepared for a reactive system, where there is a need to translate several rules and axioms during the life time of the system.

The original **translator** receives the ontology O and a set of non-monotonic rules NM. The ELK reasoner is used to classify the ontology and returns to the **translator** the inferred axioms O^+. Then, the inferred axioms together with the ontology are translated, followed by the set of non-monotonic rules. The final result of the translation process is a set of rules ($ORules \cup NMRules$), which are sent to the XSB engine. The pseudo-code for this initial procedure is shown in Algorithm 9.

Algorithm 9 Initial procedure to produce the hybrid knowledge base

procedure INITIALTRANSLATION(O, NM)
  $O^+ \leftarrow getInferredAxioms(O)$
  $ORules \leftarrow translateAxioms(O \cup O^+)$
  $NMRules \leftarrow translateRules(NM)$
return $ORules \cup NMRules$
end procedure

This initial procedure was maintained in our **translator** version. We needed a **translator** that was not only capable of translating a whole ontology and a set of non-monotonic rules, but also capable of translating a single axiom or a non-monotonic rule, maintaining always the compatibility with what was previously translated.

The first step towards the modification of this component was to understand how it was developed. Without this initial step it would be impossible for us to take advantage of what was already developed and thus speed up the **translator** development.

However, some difficulties arose during this initial phase, mostly due to the lack of

the documentation of the **translator** implementation. The process of translating an ontology and a set of non-monotonic rules into an hybrid knowledge base was out of the scope of this thesis, and is described in [IKL13].

As explained, when the **controller** receives an action to be translated it has to send the action to the **translator**. The process of translating an axiom is different for each kind of action. Also, translating an axiom requires some additional steps before it is actually translated. These required steps are necessary because the hybrid knowledge base also considers the inferred axioms of the ontology. These inferred axioms represent the knowledge that can be inferred from the set of axioms that compose an ontology. Thus, when the ontology changes, it is necessary to compute the inferred axioms to see if there is new knowledge.

Algorithms 10 and 11 show the pseudo-code of the procedures to translate an axiom.

Algorithm 10 Procedure to translate an axiom to be inserted

procedure TRANSLATEAXINSERTION($Axiom$,O)
$O^+ \leftarrow getInferredAxioms(O)$
$O\prime \leftarrow addAxiom(O, Axiom)$
$O\prime^+ \leftarrow getInferredAxioms(O\prime)$
$InfAxs \leftarrow O\prime^+ \setminus O^+$
$Axs \leftarrow InfAxs \cup Axiom$
$Rules \leftarrow translateAxioms(Axs)$
return $Rules$
end procedure

Since the reasoner only gives the whole set of inferred axioms, it is necessary to compute the difference between the set of inferred axioms before the ontology is modified (O^+) and the set of inferred axioms after the ontology is modified ($O\prime^+$). Then, the set resulting from this difference is translated along with the axiom to be inserted. The resulting rules $Rules$ are then returned to the **controller** to be sent to the *reactive system* to be inserted into the hybrid knowledge base.

Algorithm 11 Procedure to translate an axiom to be removed

procedure TRANSLATEAXREMOVAL($Axiom$,O)
$O^+ \leftarrow getInferredAxioms(O)$
$O\prime \leftarrow removeAxiom(O, Axiom)$
$O\prime^+ \leftarrow getInferredAxiom(O\prime)$
$InfAxs \leftarrow O^+ \setminus O\prime^+$
$Axs \leftarrow InfAxs \cup Axiom$
$Rules \leftarrow translateAxioms(Axs)$
return $Rules$
end procedure

This procedure is similar to the previous one. However there is a small difference. In this algorithm the order of the sets, in the set difference operation, is changed. Since

an axiom is being removed from the ontology, making the ontology smaller in terms of number of axioms, it is expected that the new set of inferred axioms is smaller than the older one, in terms of number of axioms. Also, the resulting rules $Rules$ are returned to the **controller** to be sent to the *reactive system* to be retract from the hybrid knowledge base.

It is important to point out that the reasoner used in this project, the ELK reasoner, has an interesting feature called *incremental reasoning*. This feature allows one to incrementally update the inferred class and instance hierarchies after adding, removing, or modifying axioms on classes or instances, without re-computing parts of the ontology that were not affected. This feature is used to reduce the time required to classify the ontology. It is also important to mention that, whenever an ontology becomes inconsistent, for example after inserting an axiom, the reasoner will not be able to classify the ontology. In this situation, an output message will be sent to inform the user that the ontology is inconsistent.

However, there is a particular case that is covered by the initial process, the one that produces the hybrid knowledge base, and that had to be also covered in the actions' translation process. According to [IKL13], when an ontology has a *DisjointWith* axiom the **translator** has to produce a doubled set of rules for both the ontology and the non-monotonic rules. If an ontology initially has a *DisjointWith* axiom, there is no problem when translating the actions over the ontology. However, if the initial ontology does not have such axiom, then it is necessary to check if a *DisjointWith* axiom was inserted by the action being translated. If that is the case, then it is necessary to translate the whole ontology along with the initial set of non-monotonic rules, in order to produce the double set of rules.

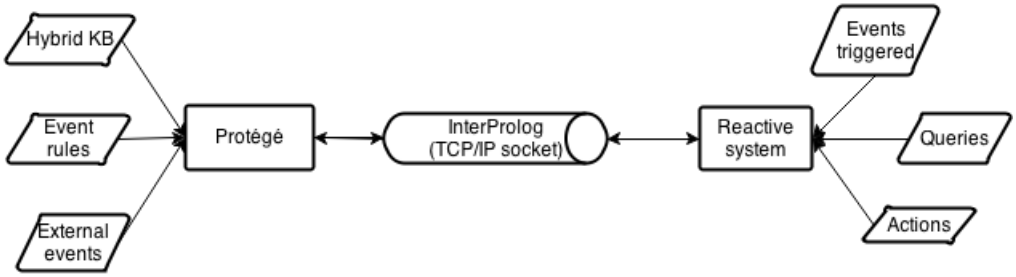
A simpler case is when the action to be translated contains a rule or a fact. In this situation, there is no need to create a different procedure for each kind of action. Also, there is no need for additional steps since the rule/facts are translated directly and returned to the **controller** to be inserted into or removed from the hybrid knowledge base.

6.4 Reactive System Wrapper

This component, the **reactive system wrapper**, was developed to support the communication between the *reactive system*, written using XSB-Prolog, and the Protégé plug-in, written using Java.

The communication between these two systems was made using InterProlog, which provides Java with the ability to call any Prolog goal through a *PrologEngine* object, and for Prolog to invoke any Java method through a *javaMessage* predicate, while passing virtually any Java objects and Prolog terms between both languages with a single instruction. In practise the communication is done through the use of TCP/IP sockets, where the objects and terms are exchanged. The scheme of what is exchanged between the systems can be seen in Figure 6.6.

Figure 6.6: Communication between Protégé plug-in and reactive system



The following example explains how to communicate between the two systems using InterProlog.

Example 14. This example shows the case when the *reactive system* sends an action to be handled by the Protégé plug-in.

```
1 notify_java_listener_insert(Term):-
2     java_obj_id(ID),
3     buildTermModel(Term,NTerm),
4     javaMessage(ID,notifyInsert(NTerm)).
```

This first piece of code, which is written in XSB-Prolog, is placed on the *reactive system* side. Whenever an action is triggered, the system calls this *notify_java_listener_insert/1* predicate, passing as argument the fact/rule to be inserted into the hybrid knowledge base.

The *buildTermModel/2* is then used to construct a Java object from the term passed as argument. The resulting object has the type TermModel, which is a representation of a term on the Java side.

After constructing the object, it is sent using the *javaMessage/2* predicate, which is used to send the object to the Java side.

```
1 public void notifyInsert(TermModel term){
2     c.handleInsert(term);
3 }
```

This last piece of code is written on the Java side. This is the method to be called by the *javaMessage* predicate whenever an insertion action has to be performed. It receives a *TermModel* object, which was constructed on the XSB-Prolog side, containing the action to be processed.

Despite presenting here only the situation where an action is sent to the plug-in, the other cases that require the communication between the two system are not much different from this one.

Using this front-end to support the communication between these two systems was not a trivial task. The lack of examples explaining how to use the API and the way the documentation was structured did not facilitate the use of it.

7

Evaluation

In this chapter we present the evaluation and the benchmarking of our *reactive system*. We decided to structure this chapter into three sections: **CEP system**, **Reactive System** and **Protégé**. In the first section we compare our CEP system with the implementation of the ETALIS system aiming to try to understand how our CEP system behaves against other Prolog engines running the ETALIS CEP system, in particular to see if our implementation achieved any improvement over the ETALIS implementation for the XSB engine. The next section is reserved to present the results for our *reactive system*, which is the result of combining the complex event rules with ECA rules. We aim to understand what is the overhead of adding the ECA rules to the complex event rules. Finally, the last section is used to show the results for our developed Protégé plug-in. The tests were performed on a Intel®Core^{TM}2 Duo @ 2.66GHz with 4GB RAM running Ubuntu 14.04 LTS OS.

7.1 CEP System

Recall that we decided to use ETALIS as a starting point to develop our CEP system. However, we noticed that their implementation was not the most efficient, at least for the XSB Prolog engine. One of our improvements over the ETALIS system was to try to make it faster when searching for EDBC rules. In this section we aim to present the results for two tests in order to see if our CEP system really got any improvement. With these tests we intend to verify three points: i) how the three engines (XSB, YAP and SWI) behave using the ETALIS implementation, ii) how our improved CEP system behaves compared with these three and iii) how our system, which is implemented using XSB, improved over the ETALIS implementation for XSB. Both tests present here were taken from the

ETALIS page[1]. We ran each test for 1000, 5000, 10000 and 25000 events, and for each set of events we ran it 3 times and took the average of the three measurements. To compare the results with our CEP system we ran the ETALIS implementation for the SWI-Prolog version 6.6.6, the YAP-Prolog 6.2.2 and XSB-Prolog version 3.4.0 (same version used for our system). To better understand the plots visualization, the *XSB*, *YAP* and *SWI* labels correspond to the ETALIS implementation whereas the label *React* corresponds to our implementation of the CEP system.

The first test aims to evaluate the computation of a transitive closure and can be seen as following:

```
1    tc(X,Y) <- r(X,Y).
2    tc(X,Z) <- tc(X,Y) seq r(Y,Z).
```

The result for this test can be seen in Figure 7.1. In the first place let us focus on point i), the comparison of the ETALIS implementation for the three engines. As one may notice their implementation for XSB has a huge difference comparing with the other two systems. For the 1000 events there is not a noticeable difference, however from the 10000 events that difference starts to become even more noticeable. What we can conclude from here is that the standard mechanism to store and find facts in the other systems are faster than XSB standard mechanism. It is also interesting to see the results for the YAP engine. Even with the 25000 events, where the other two engines exceed the 500 seconds, this engine can maintain itself in the order of 2-3 seconds, meaning that the implementation of this engine must be quite optimized. Considering now all the four systems, we can see that our CEP system (*React*) is the second fastest system. However, the most interesting point is the improvement over the ETALIS implementation for the XSB engine. As one can see, for the 25000 events, our system was 9x faster that the ETALIS implementation. With this results we can conclude that using the trie storage has an huge impact regarding the storage of facts.

The second test aims to evaluate the combination of some operators. The rules for this test can be seen as follows:

```
1    patternMatch1(Id,X,Y) <-
2        a(Id,X) seq b(Id,Y) where (Y<11).
3    conj1(Id,X,Y) <-
4        a(Id,X) and b(Id,Y) and c(Id,Z).
5    patternMatch2(Id) <- patternMatch1(Id,_,_) or conj1(Id,_,_).
```

The result for this test can be seen in Figure 7.2. Starting from point i), we can seen that *XSB* still has the worst performance and the *YAP* the best performance. Adding our CEP system, we notice a little change regarding with the performance compared with the previous test. In this scenario, the *SWI* has a better performance than our system, making it the second fastest system. Being the previous test harder than this one, we

[1] https://code.google.com/p/etalis/

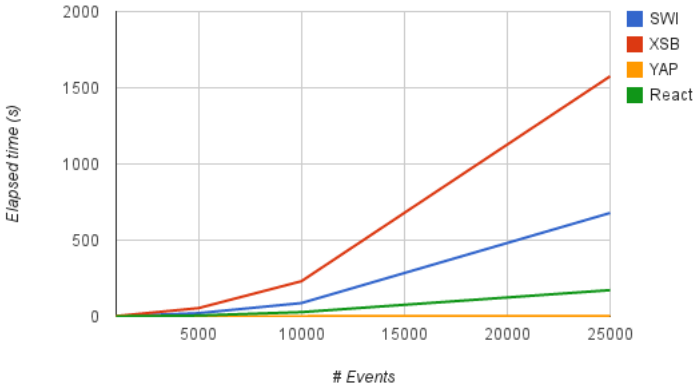


Figure 7.1: Transitive closure results

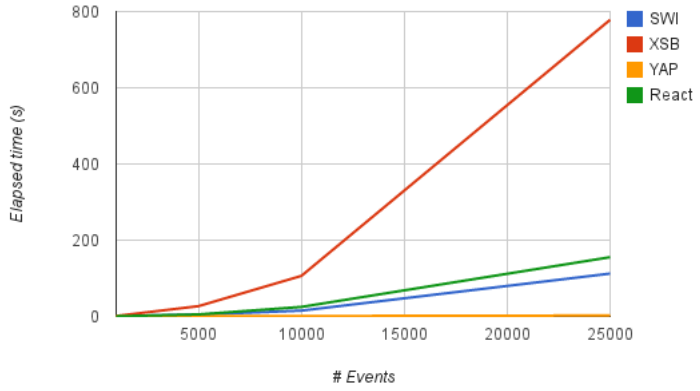


Figure 7.2: Sequence-Or-And results

believe that up to a certain point our CEP system is not better than *SWI*. However, as the computation becomes harder the *SWI* starts to decrease its performance very fast, while our system decreases its performance but not that abruptly. Even so, our system still gets a significant difference compared with the ETALIS *XSB* implementation. For the 25000 events, our system is about 5x faster than the ETALIS implementation for the XSB engine.

Our main goal with this tests was to demonstrate whether our implementation for the CEP system got any improvement over the ETALIS implementation for the XSB engine. With this results, we have shown that our implementation is in fact faster than the original ETALIS implementation.

Since using a periodic operator changes the CEP system internal architecture, we think that it is interesting to analyze what is the overhead of using three threads instead of one to process complex events. To make this comparison, we run the two previous

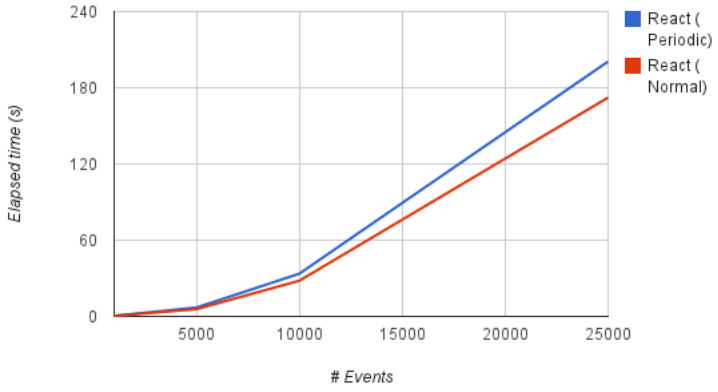


Figure 7.3: Transitive closure results for the periodic architecture

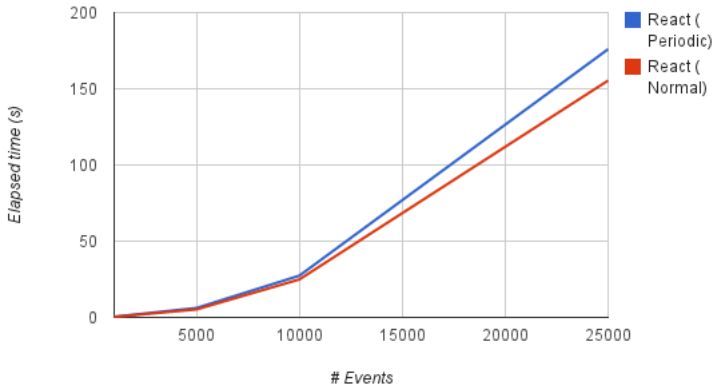


Figure 7.4: Sequence-Or-And results for the periodic architecture

tests again, but this time with the CEP system running the three threads. The results can be seen in Figures 7.3 and 7.4. Label *React (Periodic)* corresponds to the CEP system running the architecture required for the periodic, i.e, running the three threads. Label *React (Normal* corresponds to the CEP system running with only one thread.

As expected, in both tests, the CEP system running with three threads has a worse performance compared with the CEP system running with only one thread. These worse results can be explained by the extra number of instructions required to process an event, when using three threads.

7.2 Reactive System

As we explained in Chapter 5, adding the ECA rules to the CEP system became quite straightforward because we came up with a solution to transform the ECA rules into

EDBC rules. This means that, internally, the ECA rules have a similar behaviour compared with the complex event rules. In this section, we are going to show the results for only one test, which combines both types of rules. The goal of this test is just to have a notion of how the *reactive system* behaves, in terms of performance, when inserting and removing facts from the knowledge base. To get the results for this test we ran it for a set of 1000, 5000, 10000 and 25000 events, and for each set we ran it three times and we took the average of those measurements. The rules for this test are the ones as follows:

```
1  % ECA rules %
2  on a(Id,X) --> if query(not a(Id,X)) ; then insert(a(Id,X)).
3  on patternMatch1(Id,X,Y) --> if query(a(Id,X)) ; then remove(a(Id,X)).
4
5  % CE rules %
6  patternMatch1(Id,X,Y) <-
7          a(Id,X) seq b(Id,Y) where (Y<11).
8  conj1(Id,X,Y) <-
9          a(Id,X) and b(Id,Y) and c(Id,Z).
10 patternMatch2(Id) <- patternMatch1(Id,_,_) or conj1(Id,_,_).
```

The result for this test can be seen in Figure 7.5. As one might have noticed, the complex event rules of this test are the same as the complex event rules of the second test for the CEP system. The reason for using the same rules is to try to understand what is the overhead that the ECA rules bring to our system. Looking at the results, we can see that the measured times are very similar to the ones we got for the CEP system. At first glance, this result might be a little bit surprising, because there is no increase in performance when adding the ECA rules. However, these results are expected if we look at the ECA rules translation. The way we translate them, they become simple rules comparing with the translated complex event rules. We know that for each complex event rule, two or more rules are created and each of those rules execute more steps than checking a fact over the knowledge base and (possibly) an insertion or a removal from that knowledge base, as the ECA rules do. We believe that what takes more computation time is the complex event rules and that adding ECA rules will not increase significantly the computation time.

7.3 Protégé

This section is devoted to show the results of the performance tests for our Protégé plug-in. The work in [IKL13] has shown the results for the combination of the ontology with the non-monotonic rules. Those results measured the times for translating the whole ontology and the whole set of non-monotonic rules.

Here, we aim to focus on the reactive part, i.e., we aim to show how our plug-in behaves when adding and removing facts from the hybrid knowledge base. To do that,

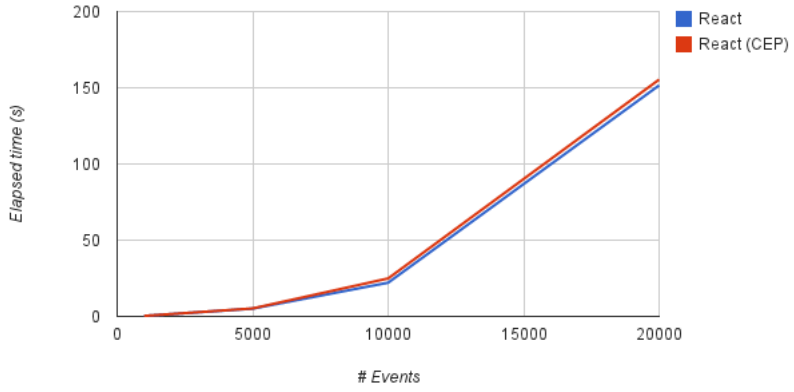


Figure 7.5: ECA and complex event rules result

we used the GALEN-OWL[2] ontology, which contains 36547 axioms, the following hand-crafted non-monotonic rules:

```
nonCardiacPhenomenon(john).
nonCardiacPhenomenon(mile).
CardiacPhenomenon(simone).

CardiacPhenomenon(X) :- CoronaryArteryDisease(X) , not
    nonCardiacPhenomenon(X).
CoronaryArteryDisease(X) :- AnginaPectoris(X).
```

along with the following set of ECA rules:

```
on ins_critical(X) --> if query(not 'Critical'(X)) ;
    then insert(classAssertion('Critical',X)).
on rem_critical(X) --> if query('VolitionalState'(X)) ;
    then remove(classAssertion('Critical',X)).
on ins_five(X) --> if query(not five(X)) ;
    then insert(classAssertion(five,X)).
on rem_five(X) --> if query(five(X)) ;
    then remove(classAssertion(five,X)).
on ins_app(X) --> if query(not 'ApplicationInformation'(X)) ;
    then insert(classAssertion('ApplicationInformation',X)).
on rem_app(X) --> if query('TopCategory'(X)) ;
    then remove(classAssertion('ApplicationInformation',X)).

on anginaDetected(X) --> if not query('AnginaPectoris'(X)) ;
    then insert('AnginaPectoris'(X)).
on checkHeartProblems(X) --> if query('CardiacPhenomenon'(X)) ;
    then insert('Critical'(X)).
on angioplastyTreatment(X) --> if query('CoronaryArteryDisease'(X)) ;
    then remove('AnginaPectoris'(X)).
```

[2] http://www.co-ode.org/galen/

Table 7.1: Processing time (ms) of each axiom insertion

	ELK	**Translation**	**XSB**	**Total**
Rule (1)	21.2	0.6	82.8	104.6
Rule (5)	12.6	0	79.6	92.2
Rule (9)	11.6	0.2	79.6	91.4

These ECA rules aim to insert and remove individuals from classes belonging to the ontology and to also insert and remove facts belonging to the set of non-monotonic rules. When writing these rules, we chose classes with different levels in the hierarchy. The queries, in some cases, are not posed to the classes where the individuals were inserted, but to classes in a higher level of the hierarchy.

To measure the performance times we generated random events, which are instances of the definitions present in the event part of the ECA rules, to make the ECA rules trigger. In each ECA rule triggered, we measured the time spent making the query and the time spent on performing the action. In the end, we took the average of both times.

The observed time spent on the queries is around 73.9ms. This time measures the duration beginning with the *reactive system* sending the query to be translated and ending when the result for this query is known. Here, it is important to point out that each predicate in the hybrid knowledge base is tabled. However, when an action occurs, all the tables created are abolished. Thus, whenever a query is posed, XSB has to create the tables again. A possible solution would be to declare the predicates as incrementally tabled and as both dynamic and incremental. However, it is not possible because XSB does currently not allow a predicate to have both declarations.

Inserting an axiom took on average 96.07ms to be executed. The time was measured starting when the *reactive system* sends the action to be translated and ending when the action has been effectively executed into the hybrid knowledge base. Table 7.1 shows the detailed processing time for each inserted axiom. The **ELK** column shows the time required to incrementally classify the ontology, after being modified. The **Translation** column shows the time required to translate the axiom being inserted along with the corresponding inferred axioms. The **XSB** column shows the time required for the plug-in to send the produced rules to the *reactive system* plus the time required to assert them into the hybrid knowledge base. Recall that, the rules are sent from Java to XSB through TCP/IP sockets, which contributes for decreasing performance for each insertion.

Looking at this table, the first thing we notice is that sending rules to be asserted is what consumes most of the processing time. The reason for this to happen is related with the way Java communicates with XSB, through TCP/IP sockets. Thus, it is possible to conclude that the time required to insert an axiom is considerably dependent on the duration of the communication process.

Regarding the removals, it takes on average 92.2ms to remove an axiom. The same phenomenon observed with the insertions can also be observed here, i.e. sending rules to be retracted is what consumes the larger part of the time due to the communication

Table 7.2: Processing time (ms) of each axiom removal

	ELK	**Translation**	**XSB**	**Total**
Rule (3)	12.2	0.2	83.4	95.8
Rule (7)	12.2	0	80.2	92.4
Rule (11)	10.2	0	78.2	88.4

process between Java and XSB. Table 7.2 shows the detailed processing time, for each rule, to remove an axiom. In this case, the **XSB** column shows the time required for the plug-in to send the produced rules plus the time XSB takes to retract them from the hybrid knowledge base.

Inserting facts (rules (14) and (16)) took, on average, 80.6ms. In this case, it is not necessary to run ELK because the rules being inserted belong to the non-monotonic part. Thus, the time required to make an insertion in this case is mostly related with the time required to send the rules to XSB to be inserted. With respect to the facts removal (rule (18)), it also took 80.6ms to execute.

7.4 Conclusions

In conclusion, we showed that changing the way ETALIS stores its facts has a significant impact on XSB performance. In some situations, our system even has a better performance then the SWI version of ETALIS. However, the YAP engine is by far the fastest one. Also, adding the reactive part to the CEP system did not have a significant negative impact on the system performance.

Regarding the Protégé plug-in, the observed execution times for only insertions and removals do not exceed one second. However, we can conclude that what consumes the majority of the processing time when inserting or removing rules is the communication process between Java and XSB, through TCP/IP sockets. To reduce this time, an extensive study and a possible optimization, which are out of the scope of this thesis, would be required.

8

Conclusion

The main purpose for this thesis was to provide a *reactive hybrid knowledge base*, whose implementation is available as a Protégé plug-in. The resulting system consists of a *reactive system* having as its background knowledge a hybrid knowledge base.

This hybrid knowledge base combines open world ontologies with closed world rule-based languages. As explained in the introductory chapter, both ontologies and rules provide distinct strengths for the representation and interchange of knowledge in the Semantic Web. Combining this knowledge with a *reactive system* allowed us to take the first step into the development of a new generation of knowledge-rich applications.

Throughout this document, we presented an overview of several works related with event algebras, complex event processing, event-condition-action rules, hybrid knowledge bases and logic-based languages and tools. Despite none of these approaches presenting something similar to what was done in this thesis, they gave us valuable insights that were then used to construct each part of the *reactive hybrid knowledge base*.

The major contributions of this work are a RIF-PRD based language to exchange ECA rules among Semantic Web engines, a *reactive system* implemented in XSB capable of modifying its background knowledge and with the ability to make complex event processing, and a Protégé plug-in to combine the *reactive system* with a hybrid knowledge base into a single framework accompanied with a translator to give support for the RIF-PRD based language.

Our most relevant contribution was the Protégé plug-in. It relies on the developed *reactive system* to give the hybrid knowledge base the desired reactivity. To form the hybrid knowledge base, the plug-in uses an adapted version of the translator used in the NoHR project.

The ETALIS system was used as basis to implement the complex event processing

part of our *reactive system*. The original implementation was modified to take advantage of the features available in XSB and also augmented with a new event operator. The reactive rules were then added to the modified system, giving rise to the *reactive system*.

Since there is no format to exchange ECA rules among Web rule engines, our language, based on RIF-PRD, was proposed to fill the gap left by the RIF working group when it comes to exchange ECA rules.

Finally, we did several performance tests to see how our solution behaved in terms of performance. Regarding the results of the performance tests, we concluded that though it is possible to have a reactive hybrid knowledge base, the final solution presented here might not be viable when inserting or removing certain kind of axioms. In some cases, the time can exceed one second to insert or remove an axiom. In contrast, the complex event processing part of our reactive system got a considerable performance improvement over the original ETALIS implementation, by changing the way facts are stored.

In future work, the system can be enhanced with an implementation for the update operator. Furthermore, new kinds of actions, e.g. an action to call external procedures, can be added to the ECA rules. With respect to the *conflict resolution*, new strategies can be implemented in the *reactive system*. The way events arrive can be also enhanced with an implementation to allow the communication through sockets between an external source and the *reactive system*. Finally, specific implementation enhancements can be performed in an attempt to reduce the time required when modifying the hybrid knowledge base with axioms.

Bibliography

[Ada05] R. Adaikkalavan. "Formalization and Detection of Events Using Interval-Based Semantics". In: *Proceedings, International Conference on Management of Data*. 2005, pp. 58–69.

[AEM09] J. J. Alferes, M. Eckert, and W. May. "Semantic Techniques for the Web". In: ed. by F. Bry and J. Maluszynski. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Evolution and Reactivity in the Semantic Web, pp. 161–200.

[AKS10] J. J. Alferes, M. Knorr, and T. Swift. "Query-driven Procedures for Hybrid MKNF Knowledge Bases". In: *CoRR* abs/1007.3515 (2010).

[All83] J. F. Allen. "Maintaining Knowledge About Temporal Intervals". In: *Commun. ACM* 26.11 (Nov. 1983), pp. 832–843.

[AFRSSS10] D. Anicic, P. Fodor, S. Rudolph, R. Stahmer, N. Stojanovic, and R. Studer. "A Rule-Based Language for Complex Event Processing and Reasoning". In: *Web Reasoning and Rule Systems*. Ed. by P. Hitzler and T. Lukasiewicz. Vol. 6333. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 42–57.

[AFRS11] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. "EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning". In: *Proceedings of the 20th International Conference on World Wide Web*. WWW '11. Hyderabad, India: ACM, 2011, pp. 635–644.

[AFSS09] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. "Event-Driven Approach for Logic-Based Complex Event Processing". In: *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*. CSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 56–63.

[ARFS12] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. "Stream Reasoning and Complex Event Processing in ETALIS". In: *Semantic Web: Interoperability, Usability, Applicability* 3.4 (Oct. 2012), pp. 397–407.

[ABW03] A. Arasu, S. Babu, and J. Widom. *The CQL Continuous Query Language: Semantic Foundations and Query Execution*. Technical Report 2003-67. Stanford InfoLab, 2003.

[BPW02] J. Bailey, A. Poulovassilis, and P. Wood. "An Event-condition-action Language for XML". In: *Proceedings of the 11th International Conference on World Wide Web*. WWW '02. Honolulu, Hawaii, USA: ACM, 2002, pp. 486–495.

[BBCVG09] D. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. "C-SPARQL: SPARQL for Continuous Querying". In: *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. Madrid, Spain: ACM, 2009, pp. 1061–1062.

[BBCVG10] D. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. "Stream Reasoning: Where We Got So Far". In: *Proceedings of the 4th International Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic (NeFoRS)*. Heraklion, Greece, May 2010.

[BLHL+01] T. Berners-Lee, J. Hendler, O. Lassila, et al. "Semantic Web". In: *Scientific american* 284.5 (2001), pp. 28–37.

[BBBEP07] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P. Patranjan. "Reactive Rules on the Web". In: *Reasoning Web*. Vol. 4636. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 183–239.

[BK13] H. Boley and M. Kifer. *RIF Basic Logic Dialect*. Feb. 2013. URL: http://www.w3.org/TR/2010/REC-rif-bld-20100622/.

[BKHPPR13] H. Boley, M. Kifer, G. Hallmark, A. Paschke, A. Polleres, and D. Reynolds. *RIF Core Dialect*. Feb. 2013. URL: http://www.w3.org/TR/2013/REC-rif-core-20130205/.

[BK96] A. J. Bonner and M. Kifer. "Concurrency and Communication in Transaction Logic". In: *Joint Intl. Conference and symposium on logic programming*. MIT Press, 1996, pp. 142–156.

[Bra07] S. Bratt. *Semantic Web, and Other Technologies to Watch*. Jan. 2007. URL: http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/.

[BG14] D. Brickley and R. Guha. *RDF Schema 1.1*. Feb. 2014. URL: http://www.w3.org/TR/rdf-schema/.

[CL04] J. Carlson and B. Lisper. "An Event Detection Algebra for Reactive Systems". In: *Proc. 4TH ACM Int. Conference on embedded software (EMSOFT)*. ACM, 2004, pp. 147–154.

[CM94] S. Chakravarthy and D. Mishra. "Snoop: An Expressive Event Specification Language for Active Databases". In: *Data Knowl. Eng.* 14.1 (Nov. 1994), pp. 1–26.

[CCDFHHKMRS03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. "TelegraphCQ: Continuous Dataflow Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 668–668.

[CW96] W. Chen and D. S. Warren. "Tabled Evaluation with Delaying for General Logic Programs". In: *J. ACM* 43.1 (Jan. 1996), pp. 20–74.

[DAL10] C. Damásio, J. Alferes, and J. Leite. "Declarative Semantics for the Rule Interchange Format Production Rule Dialect". In: *The Semantic Web ISWC 2010*. Ed. by P. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Pan, I. Horrocks, and B. Glimm. Vol. 6496. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 798–813.

[DM07] W. Drabent and J. Maluszynski. "Well-founded Semantics for Hybrid Rules". In: *Proceedings of the 1st International Conference on Web Reasoning and Rule Systems*. RR'07. Innsbruck, Austria: Springer-Verlag, 2007, pp. 1–15.

[EB10] M. Eckert and F. Bry. "Rule-based Composite Event Queries: The Language XChangeEQ and Its Semantics". In: *Knowl. Inf. Syst.* 25.3 (Dec. 2010), pp. 551–573.

[EILST08] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. "Combining answer set programming with description logics for the Semantic Web". In: *Artificial Intelligence* 172 (2008), pp. 1495 – 1539.

[GSS04] D. Goldin, S. Srinivasa, and V. Srikanti. "Active databases as information systems". In: *Database Engineering and Applications Symposium, 2004. IDEAS '04. Proceedings. International*. 2004, pp. 123–130.

[Har87] D. Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231 –274.

[HKPPSR12] P. Hitzler, M. Krotzsch, B. Parsia, P. Patel-Schneider, and S. Rudolph. *OWL 2 Web Ontology Language Primer*. Dec. 2012. URL: http://www.w3.org/TR/owl2-primer/.

[HB11] M. Horridge and S. Bechhofer. "The OWL API: A Java API for OWL Ontologies". In: *Semant. web* 2.1 (Jan. 2011), pp. 11–21.

[IKL13] V. Ivanov, M. Knorr, and J. Leite. "A Query Tool for $\mathcal{EL}$ with Non-monotonic Rules". In: *The Semantic Web ISWC 2013*. Ed. by H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J. Parreira, L. Aroyo, N. Noy, C. Welty, and K. Janowicz. Vol. 8218. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 216–231.

[KKS12] Y. Kazakov, M. Krotzsch, and F. Simancik. *ELK: A Reasoner for OWL EL Ontologies*. System Description. University of Oxford, 2012.

[KCM14] G. Klyne, J. Carroll, and B. McBride. *RDF 1.1 Concepts and Abstract Syntax*. Feb. 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/Overview.html.

[KAH11] M. Knorr, J. J. Alferes, and P. Hitzler. "Local closed world reasoning with description logics under the well-founded semantics". In: *Artificial Intelligence* 175 (2011), pp. 1528 –1554.

[Lif91] V. Lifschitz. "Nonmonotonic Databases and Epistemic Queries". In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'91. Sydney, New South Wales, Australia: Morgan Kaufmann Publishers Inc., 1991, pp. 381–386.

[MPSP12] B. Motik, P. F. Pater-Schneider, and B. Parsia. *OWL 2 Web Ontology Language Structural Specification and Functional-style Syntax*. Dec. 2012. URL: http://www.w3.org/TR/owl2-syntax.

[MR10] B. Motik and R. Rosati. "Reconciling Description Logics and Rules". In: *J. ACM* 57.5 (June 2010), 30:1–30:62.

[PPW04] G. Papamarkos, R. Poulovassilis, and P. Wood. "RDFTL: An Event-Condition-Action Language for RDF". In: *In Proc. 3rd Int. Workshop on Web Dynamics (in conjunction with WWW2004)*. 2004, pp. 223–248.

[Pas06] A. Paschke. "ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics". In: *CoRR* abs/cs/0610167 (2006).

[PBK13] A. Polleres, H. Boley, and M. Kifer. *RIF Datatypes and Built-Ins*. Feb. 2013. URL: http://www.w3.org/TR/2013/REC-rif-dtb-20130205.

[SMHP13] C. de Sainte Marie, G. Hallmark, and A. Paschke. *RIF Production Rule Dialect*. Feb. 2013. URL: http://www.w3.org/TR/2010/REC-rif-bld-20100622/.

[SW13] T. Swift and D. Warren. *The XSB Programmer's Manual: Version 3.3.x volume 1*. 2013.

[WJFY08] O. Walavalkar, A. Joshi, T. Finin, and Y. Yesha. "Streaming knowledge bases". In: *In International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2008.

Appendix: Supported Language

We start by presenting the syntax for the proposed language. To make the presentation of the syntax easier to read, we split the presentation into three parts: the event language, the condition language and the action language. Each language deals with a specific part of the whole language. For the sake of readability and simplicity, this document also introduces a notation that is not intended to be the concrete syntax, since the only concrete syntax is the XML syntax.

A.1 Event Language

This section specifies the syntax of the event language, which deals with the event part of the ECA rules. Besides that, it also introduces the notion of *patterns* which will be used to construct complex event rules.

A.1.1 Alphabet

Definition 4. The **alphabet** of the language consists of:

- set of **constants symbols** $Const$.
- set of **variable symbols** *Var* (disjoint from $Const$).
- syntactic constructs to denote:
  - connective symbols: *EAnd, EOr, EPar, ESeq, EEquals, EMeets, EDuring, EStart, EFinishes, EPeriodic and ENot.*

A.1.2 Terms

The most basic construct in the event language is the term. There are two kinds of terms: *constants* and *variables*. *Variables* are represented with a "?" followed by the name of the variable.

Definition 5. Term. If $t \in Const$ or $t \in Var$ then t is a **term**.

A.1.3 Formulas

Definition 6. Atomic event formula. If P_e is an n-ary predicate symbol and $t_1 \dots t_n$ are terms then $P_e(t_1 \dots t_n)$ is an atomic event formula.

Definition 7. Event formula. An event formula can have several different forms and is defined as follows:

- **Event atom**: If φ is an atomic event formula then it is also an event formula.
- **Patterns**: A pattern is either an atomic event formula or an event formula that has one of the following forms:
  - *Conjunction*: if φ_1 and φ_2 are patterns then so is `EAnd(`φ_1 φ_2`)`.
  - *Disjunction*: if φ_1 and φ_2 are patterns then so is `EOr(`φ_1 φ_2`)`.
  - *Concurrent Conjunction*: if φ_1 and φ_2 are patterns then so is `EPar(`φ_1 φ_2`)`.
  - *Sequence*: if φ_1 and φ_2 are patterns then so is `ESeq(`φ_1 φ_2`)`.
  - *Equals*: if φ_1 and φ_2 are patterns then so is `EEquals(`φ_1 φ_2`)`.
  - *Meets*: if φ_1 and φ_2 are patterns then so is `EMeets(`φ_1 φ_2`)`.
  - *During*: if φ_1 and φ_2 are patterns then so is `EDuring(`φ_1 φ_2`)`.
  - *Start*: if φ_1 and φ_2 are patterns then so is `EStart(`φ_1 φ_2`)`.
  - *Finishes*: if φ_1 and φ_2 are patterns then so is `EFinishes(`φ_1 φ_2`)`.
  - *Periodic*:
    - if $t \in Const$, representing a temporal step, and $t_1, t_2 \in Const$, representing an initial time and a final time, then `EPeriodic(`t_1 t t_2`)` is a pattern.
    - if $t \in Const$, representing a temporal step, and φ_1, φ_2 are patterns, then `EPeriodic(`φ_1 t φ_2`)` is a pattern.
  - *Not*: if φ_1 and φ_2 are patterns, then so is `ENot(`φ_1 φ_2`)`.

A.1.4 Example

Example 15. When some employee leaves the work before the working day ends, an event should be raised. This situation can be described using the following event formula:

```
EDuring(ex:employee_left(?P) ex:working_day())
```

The "ex:" before the names of the events represents an abbreviated syntax for expressing a fictitious URI ("http://www.example.com"). Throughout the following examples, the same abbreviated syntax will be used.

A.2 Condition Language

This section specifies the syntax of the condition language, which deals with the condition part of the ECA rules.

A.2.1 Alphabet

Definition 8. The **alphabet** of the condition language consists of:

- a countably infinite set of constant symbols $Const$ and variable symbols Var (same sets as the ones described earlier in the event language),
- and syntactic constructs (same of RIF-PRD) to denote:
  - lists,
  - relations, including equality, class membership and subclass relations
  - conjunction, disjunction and negation,
  - and existential conditions.

A.2.2 Terms

The most basic construct in the condition language is the term. Several kinds of terms are defined, such as: *constants, variables, positional terms and lists*.

Definition 9. Term. This definition extends Definition 5 with the following constructs:

- **List terms.** A list has the form `List(`$t_1 \ldots t_m$`)`, where $m \geq 0$ and $t_1, \ldots, t_m$ are ground terms, i.e. without variables. A list of the form `List()` is called empty list.
- **Positional terms.** If $t \in Const$ and $t_1, \ldots, t_n,\ n \geq 0$, are terms then $t(t_1 \ldots t_n)$ is a positional term. The constant t represents a function and $t_1, \ldots, t_n$ represent argument values.

A.2.3 Formulas

Definition 10. Atomic condition formula. An atomic condition formula can have several different forms and is defined as follows:

- **Positional atomic formulas:** If P is an n-ary predicate symbol and $t_1, \ldots, t_n, n \geq 0$, are terms then $P(t_1, \ldots, t_n)$ is a positional atomic formula.

- **Equality atomic formulas:** $t = s$ is an equality atomic formula, if t and s are terms.
- **Class membership atomic formulas:** $t\#s$ is a membership atomic formula, if t and s are terms. The term t is the object and the term s is the class.
- **Subclass atomic formulas:** $t\#\#s$ is a subclass atomic formula if t and s are terms.
- **Frame atomic formula:** $t[p_1 \rightarrow v_1 \ldots p_n \rightarrow v_n]$ is a frame atomic formula if $t, p_1, \ldots, p_n, v_1, \ldots, v_n, n \geq 0$, are terms. The term t is the object of the frame; the p_i are the property or attribute names; and the v_i are the property or attribute values.

Definition 11. Condition formula.

- **Condition atom:** If ϕ is an atomic condition formula then it is also a condition formula.
- **Conjunction:** If $\phi_1, \ldots, \phi_n, n \geq 0$, are condition formulas then so is `And(`$\phi_1 \ldots \phi_n$`)`, called a conjunctive formula. As a special case, `And()` is allowed and is treated as a tautology.
- **Disjunction:** $\phi_1, \ldots, \phi_n, n \geq 0$, are condition formulas then so is `Or(`$\phi_1 \ldots \phi_n$`)`, called a disjunctive formula. As a special case, `Or()` is allowed and is treated as a contradiction.
- **Negation:** If ϕ is a condition formula, then so is `Not(`ϕ`)`, called negative formula.
- **Existential:** If ϕ is a condition formula and $?V_1, \ldots, ?V_n, n \geq 0$, are variables then `Exists` $?V_1 \ldots ?V_n(\phi)$ is an existential formula.

A.2.4 Examples

Example 16. A condition formula that checks if an employee is an area manager can be represented as follows:

```
And(ex:employee(?P) ex:area_manager(?P))
```

A.3 Action Language

This section defines the syntax of the action language, which deals with the action part of the ECA rules.

A.3.1 Alphabet

Definition 12. The **alphabet** of the action language includes symbols to denote:

- the assertion of a fact represented by a condition atom,
- the retraction of a fact represented by a condition atom,
- the update of a fact represented by a condition atom.

A.3.2 Actions

Definition 13. Action.

1. **Assert fact:** If ϕ is a condition atom in the condition language, then `Assert(`ϕ`)` is an action.
2. **Retract fact:** If ϕ is a condition atom in the condition language, then `Retract(`ϕ`)` is an action.
3. **Update fact:** If ϕ is a condition atom in the condition language, then `Update(`ϕ`)` is an action.

A.3.3 Action Blocks

The action block is the top level construct to represent the conclusions of the rules. An action block contains a non-empty sequence of actions.

Definition 14. Action block. If $a_1, \ldots, a_n, n \geq 1$, are actions, then `Do(`$a_1 \ldots a_n$`)` denotes an action block.

A.3.4 Example

Example 17. The action of firing employees A and B can be represented as follows:

```
Do(Retract(ex:employee("A")) Retract(ex:employee("B")))
```

A.4 Rules

A.4.1 Abstract Syntax

The alphabet of the rule language includes the alphabets of the event, condition and the action language and adds symbols for:

- combining event atoms and patterns into a rule,
- combining an event, a condition and an action block into a rule,
- grouping rules.

A.4.2 Rules

Definition 15. Rule. A rule can be:

1. **Event-condition-action rule**: if `event` is an event atom in the event language, if `condition` is a formula in the condition language, and if `action` is an action block, then `On event If condition Then action` is an event-condition-action rule.

2. **Event rule**: If `complex event` is an event atom and if `pattern` is an event pattern then `complex event <- pattern` is an event rule.

Definition 16. Group. A group consists of a, possibly empty, set of rules and groups, associated with a conflict resolution strategy. A group can be:

1. Event rule group: If $\varphi_1, \ldots, \varphi_n, n \geq 0$, are event rules or event rule groups, then `EGroup(`$\varphi_1 \ldots \varphi_n$`)` is an event rule group.

2. ECA rule group: If $\psi_1, \ldots, \psi_n, n \geq 0$, are event-condition-action rules or ECA rule groups, then `ECAGroup(`$\psi_1 \ldots \psi_n$`)` is an ECA rule group.

Definition 17. Document. A document consists of a zero or more groups.

A.5 Presentation Syntax

To make it easier to read, a non-normative, lightweight notation was introduced to complement the abstract syntax. The presentation syntax is not normative. The EBNF for the presentation syntax is given as follows, where the BNF-style conventions for elements are used: "?" denotes optionality, "*" denotes zero or more occurrences, "+" denotes one or more occurrences.

Document :

⟨*Document*⟩ ::= `Document (`⟨*Prefix*⟩`*` ⟨*EGroup*⟩`?` ⟨*ECAGroup*⟩`?)`

⟨*Prefix*⟩ ::= `Prefix (`⟨*Name*⟩ ⟨*ANGLEBRACKIRI*⟩`)`

Rule language :

⟨*EGroup*⟩ ::= `EGroup ((`⟨*Event_rule*⟩ | ⟨*EGroup*⟩`)*)`

⟨*ECAGroup*⟩ ::= `ECAGroup ((`⟨*ECA_rule*⟩ | ⟨*ECAGroup*⟩`)*)`

⟨*Event_rule*⟩ ::= ⟨*Event_atom*⟩ <– ⟨*Pattern*⟩

⟨*ECA_rule*⟩ ::= `On (`⟨*Event_atom*⟩`) If (`⟨*Condition*⟩`) Then (`⟨*Action_block*⟩`)`

Action language :

⟨*Action_block*⟩ ::= `Do (`⟨*Action*⟩`+)`

⟨*Action*⟩ ::= `Assert (`⟨*Atomic*⟩`)`
| `Retract (`⟨*Atomic*⟩`)`
| `Update (`⟨*Atomic*⟩`)`

Condition language :

⟨*Condition*⟩ ::= ⟨*Atomic*⟩
| `And` (⟨*Condition*⟩*)
| `Or` (⟨*Condition*⟩*)
| `Not` (⟨*Condition*⟩)
| `Exists` (⟨*Var*⟩+) (⟨*Condition*⟩)

⟨*Atomic*⟩ ::= (⟨*Atom*⟩ | ⟨*Frame*⟩)

⟨*Atom*⟩ ::= ⟨*Const*⟩ (⟨*Term*⟩*)

⟨*Frame*⟩ ::= ⟨*Term*⟩ [(⟨*Term*⟩ -> ⟨*Term*⟩)*]

Event language :

⟨*Event_atom*⟩ ::= ⟨*Const*⟩ (⟨*Term*⟩*)

⟨*Pattern*⟩ ::= ⟨*Event_atom*⟩
| ⟨*Op*⟩ (⟨*Pattern*⟩ ⟨*Pattern*⟩)
| `EPeriodic` (⟨*Const*⟩ ⟨*Const*⟩ ⟨*Const*⟩)
| `EPeriodic` (⟨*Pattern*⟩ ⟨*Const*⟩ ⟨*Pattern*⟩)

⟨*Op*⟩ ::= `ESeq`
| `EAnd`
| `EOr`
| `EPar`
| `EMeets`
| `EDuring`
| `EEquals`
| `EStart`
| `EFinishes`
| `ENot`

⟨*Term*⟩ ::= (⟨*Const*⟩ | ⟨*Var*⟩ | ⟨*List*⟩)

⟨*Const*⟩ ::= ⟨*UNICODESTRING*⟩^^⟨*SYMSPACE*⟩
| ⟨*CONSTSHORT*⟩

⟨*Var*⟩ ::= `?` ⟨*Name*⟩

⟨*List*⟩ ::= `List` ((⟨*Const*⟩ | ⟨*List*⟩)*)

⟨*SYMSPACE*⟩ ::= ⟨*ANGLEBRACKIRI*⟩
| ⟨*CURIE*⟩

ANGLEBRACKIRI is used to represent IRIs written inside angle brackets, *UNICODESTRING* is a Unicode string, *CONSTSHORT* is used to represent several ways to express constants and *CURIE* is used to abbreviate symbol space IRIs. The document [PBK13] gives further information about these symbols.

A.6 Example

In this section we present an example that makes use of both ECA rules and complex event rules.

Example 18. When some employee leaves the work before the working day ends, an event should be raised. If that employee is a new employee then he/she should be put in some watch list. This situation can be described as follows:

```
EGroup(
ex:employee_warn(?E) <- ex:employee_left(?E) during ex:working_day()
)
ECAGroup(
On ex:employee_warn(?E)
If (And(ex:employee(?E) ex:status(?E "new")))
Then Do(Assert(ex:watchlist(?E)))
)
```

A.7 XML Syntax

The XML syntax of the RIF language is specified for each component as a pseudo-schema, as part of the description of the component. The pseudo-schema use BNF-style conventions for attributes and elements: "?" denotes optionality, "*" denotes zero or more occurrences, "+" denotes one or more occurrences, "[" and "]" are used to form groups, and "|" represents choice.

A.7.1 Events

This section specifies the XML constructs that are used to serialize event formulas.

A.7.1.1 Terms

Const The `Const` element is used to serialize a constant. It has a required attribute `type`. The content of a `Const` element can be any Unicode character string.

```
<Const type=rif:iri>
  Any string here
</Const>
```

Var The `Var` element is used to serialize a variable.

```
<Var>xsd:NCName</Var>
```

A.7.1.2 Event Formulas

Event atom The element `EAtom` is used to serialize an atomic event formula.

```
<EAtom>
  <op> Const <op>
  (<args> Terms+ </args>)?
</EAtom>
```

Conjunction The element `EAnd` is used to serialize a conjunction pattern.

```
<EAnd>
  <pattern> Event Formula </pattern>
  <pattern> Event Formula </pattern>
</EAnd>
```

Disjunction The element `EOr` is used to serialize a disjunction pattern.

```
<EOr>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EOr>
```

Concurrent conjunction The element `EPar` is used to serialize a concurrent conjunction pattern.

```
<EPar>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EPar>
```

Sequence The element `ESeq` is used to serialize a conjunction pattern.

```
<ESeq>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</ESeq>
```

Equals The element `EEquals` is used to serialize a equals pattern.

```
<EEquals>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EEquals>
```

Meets The element `EMeets` is used to serialize a meets pattern.

```
<EMeets>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EMeets>
```

During The element `EDuring` is used to serialize a during pattern.

```
<EDuring>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EDuring>
```

Start The element `EStart` is used to serialize a start pattern.

```
<EStart>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EStart>
```

Finishes The element `EFinishes` is used to serialize a finishes pattern.

```
<EFinishes>
  <pattern> Event Formula <pattern>
  <pattern> Event Formula <pattern>
</EFinishes>
```

Periodic The element `EPeriodic` is used to serialize a periodic pattern.

```
<EPeriodic>
  <start> Const </start>
  <step> Const </step>
  <end>  Const </end>
</EPerodic>
```

```
<EPeriodic>
  <start> Event Formula </start>
  <step> Const </step>
  <end>  Event Formula </end>
</EPerodic>
```

Not The element `ENot` is used to serialize the negation pattern.

```
<ENot>
    <pattern> Event Formula </pattern>
    <pattern> Event Formula </pattern>
</ENot>
```

A.7.2 Conditions

A.7.2.1 Terms

List The element `List` is used to serialize a list. This element contains an optional `items` element that contains one or more terms without variables that serialize the elements of the list.

```
<List>
  (<items> Terms+ </items>)?
</List>
```

A.7.2.2 Condition Formulas

Positional atomic formula The `Atom` element is used to serialize a position atomic formula.

```
<Atom>
  <op>Const</op>
  <args> Term+ </args>
</Atom>
```

Equal The `Equal` element is used to serialize equality atomic formulas. This element must contain one `left` sub-element and one `right` sub-element. The content of these two sub-elements must be a term.

```
<Equal>
  <left> Term </left>
  <right> Term </right>
</Equal>
```

Member The `Member` element is used to serialize membership atomic formulas. This element must contains a sub-element `instance` that serializes the reference to the object and a sub-element `class` that serializes the reference to the class.

```
<Member>
  <instance> Term </instance>
  <class> Term </class>
</Member>
```

Subclass The `Subclass` element is used to serialize subclass atomic formulas. This element must contain a sub-element `sub` that serializes the reference to the subclass and a sub-element `super` that serializes the reference to the super-class.

```
<Subclass>
  <sub> Term </sub>
  <super> Term </super>
<Subclass>
```

Frame The `Frame` element is used to serialize frame atomic formulas. This element must contain a sub-element `object` that serializes the reference to the frame's object and zero or more sub-element `slot` each serializing an attribute-value pair.

```
<Frame>
  <object> Term </object>
  (<slot> Term Term</slot>)*
</Frame>
```

And The `And` element is used to serialize a conjunction. This element contains zero or more `formula` sub-elements, each containing a formula.

```
<And>
  (<formula> Condition Formula </formula>)*
</And>
```

Or The `Or` element is used to serialize a disjunction. This element contains zero or more `formula` sub-elements, each containing a formula.

```
<Or>
  (<formula> Condition Formula </formula>)*
</Or>
```

Neg The `Neg` element is used to serialize the negation. This element contains only one `formula` sub-element.

```
<Neg>
  <formula> Condition Formula </formula>
</Neg>
```

Exists The `Exists` element is used to serialize an existentially quantified formula. This element must contain one or more `declare` sub-elements, each containing one variable; and exactly one `formula` sub-element.

```
<Exists>
  (<declare> Var </declare>)+
  <formula> Condition Formula </formula>
</Exists>
```

A.7.3 Actions

A.7.3.1 Actions

Assert The `Assert` element is used to serialize an assertion action. This element contains one `target` sub-element containing an `Atom` element.

```
<Assert>
  <target> Atom </target>
</Assert>
```

Retract The `Retract` element is used to serialize an retract action. This element contains one `target` sub-element containing an `Atom` element.

```
<Retract>
  <target> Atom </target>
</Retract>
```

Update The `Update` element is used to serialize an update action. This element contains one `target` sub-element containing an `Atom` element.

```
<Update>
  <target> Atom </target>
</Update>
```

A.7.3.2 Action Blocks

Action block The `Do` element is used to serialize an action block. This element must contain a `actions` sub-element containing a sequence of one or more actions.

```
<Do>
  <actions> Action+ </actions>
</Do>
```

A.7.4 Rules

A.7.4.1 Rules

Event-condition-action rule The `ECARule` is used to serialize an event-condition-action rule. This element contains one `on` sub-element containing an event atom, one `if` sub-element containing a condition formula and one `then` sub-element containing an action.

```
<ECARule>
  <on> Event atom </on>
  <if> Condition formula</if>
  <then> Action </then>
</ECARule>
```

Event rule The `ERule` element is used to serialize an event rule. This element contains one `head` sub-element containing an event atom and one `body` sub-element containing a pattern.

```
<ERule>
  <head> Event atom </head>
  <body> Pattern </body>
</ERule>
```

A.7.4.2 Groups

Event rule group The `EGroup` element is used to serialize a event rule group. This element contains zero or more `esentence` sub-elements each containing a event rule or a event rule group.

```
<EGroup>
  <esentence> [ Event rule | Event rule group ] </esentence>
</EGroup>
```

ECA rule group The `ECAGroup` element is used to serialize an ECA rule group. This element contains zero or more `ecasentence` sub-elements, each containing a event-condition-action rule or a ECA rule group.

```
<ECAGroup>
  <ecasentence> [ ECA rule | ECA rule group ] </ecasentence>
</ECAGroup>
```

A.7.4.3 Document

Document The `Document` element is the root element of the any RIF-ECARD document. It contains a zero or one *payload* sub-element that must contain zero or one *ECA rule group* and zero or one *Event rule group* element.

```
<Document>
    <payload>
        ECAGroup?
        EventGroup?
    </payload>?
</Document>
```

A.7.4.4 Example

The serialization of the example described in 18 can be seen in Appendix B.

As mentioned, the **event language** was our starting point. It was completely designed from scratch and nothing presented there is compatible with the RIF-PRD dialect.

The remaining main subsets of the language (the condition and action language) were mostly inspired from the RIF-PRD. In particular, we decided to keep the condition language as it is presented on the RIF-PRD. The action language, on the other side, has not full compatibility with what is presented on RIF-PRD. We decided to remove the possibility to retract all the slot values, to retract objects and to execute external actions. Also, the compound actions were removed.

The syntax of the rules had also to be modified compared with the basis dialect, because being the original dialect indicated for production rules, the event part was missing. Since our language has support for the complex event rules, a new syntax had to be

created for those rules.

In the original dialect only one type of group was allowed. However, since our dialect supports two kinds of rules, the ECA rules and the complex event rules, it was necessary to add an extra group for the latter kind of rules.

The only part of our language that is fully compatible with the RIF-PRD, at the syntax level, is the **condition language** and the **action language**. All the other language constructions were modified to adapt to the needs required by the ECA rules, meaning that they are no longer compatible with the RIF-PRD.

In contrast, the only part of RIF-PRD that is fully compatible with our language is the **condition language**. All the other constructions of RIF-PRD language are not supported by our language.

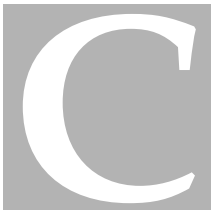
B

Appendix: XML Complete Example

```
<Document>
    <payload>
        <EGroup>
            <esentence>
                <ERule>
                    <head>
                        <EAtom>
                            <op>
                                <Const type="rif:iri">ex:employee_warn</Const>
                            </op>
                            <args><Var> ?e </Var></args>
                        </EAtom>
                    <body>
                        <EDuring>
                            <pattern>
                                <EAtom>
                                    <op>
                                        <Const type="rif:iri">
                                        ex:employee_left
                                        </Const>
                                    </op>
                                    <args>
                                        <Var> ?e </Var>
                                    </args>
                                </EAtom>
                            </pattern>
                            <pattern>
                                <EAtom>
                                    <op>
```

```
                                    <Const type="rif:iri">
                                    ex:working_day
                                    </Const>
                                </op>
                            </EAtom>
                        </pattern>
                    </EDuring>
                </body>
            </ERule>
        </esentence>
    </EGroup>
    <ECAGroup>
        <ECARule>
        <ecasentence>
            <on>
                <EAtom>
                    <op>
                        <Const type="rif:iri">ex:employee_warn</Const>
                    </op>
                    <args>
                        <Var> ?e </Var>
                    </args>
                </EAtom>
            </on>
            <if>
                <And>
                    <formula>
                        <Atomic>
                            <op>
                                <Const type="rif:iri">
                                ex:employee
                                </Const>
                            </op>
                            <args>
                                <Var>?e</Var>
                            </args>
                        </Atomic>
                    </formula>
                    <formula>
                        <Atomic>
                            <op>
                                <Const type="rif:iri">
                                ex:status
                                </Const>
                            </op>
                            <args>
                                <Var>?e</Var>
                                <Const type="xsd:string">
                                new
                                </Const>
```

```
                                    </args>
                                </Atomic>
                            </formula>
                        </And>
                    </if>
                    <then>
                        <Do>
                            <actions>
                                <Assert>
                                    <target>
                                        <Atom>
                                            <op>
                                                <Const type="rif:iri">
                                                ex:watchlist
                                                </Const>
                                            </op>
                                            <args>
                                                <Var>?e</Var>
                                            </args>
                                        </Atom>
                                    </target>
                                </Assert>
                            </actions>
                        </Do>
                    </then>
                <ecasentence>
                </ECARule>
            </ECAGroup>
        <payload>
<Document>
```



Appendix: Supported Axioms

Here is presented the axioms supported by our Protégé plug-in. The axioms syntax is similar to the OWL functional syntax. However, since the axioms are used in the ECA rules actions, the OWL functional syntax had to be modified to be compatible with the XSB syntax.

Section C.1 shows all the available axioms that can be inserted or retracted from the hybrid knowledge base. The remained sections show the auxiliary elements used to construct the axioms. Throughout the chapter, the following symbols are used to help describing the elements:

- <*Elem*> , ... , <*Elem*> is used to represent a list of two or more <*Elem*> elements.
- <*Elem*> , ... is used to represent a list of one or more *Elem* elements.
- *[<Elem>]* is used when <*Elem*> element is optional.

For further information about the meaning of each axiom the reader is referred to [MPSP12].

C.1 Axioms

C.1.1 Class Expression Axiom

C.1.1.1 Subclass Axiom

$subClassOf(< subClassExpression >, < superClassExpression >)$

⟨*subClassExpression*⟩ ::= ⟨*ClassExpression*⟩

⟨*superClassExpression*⟩ ::= ⟨*ClassExpression*⟩

C.1.1.2 Equivalent Classes Axiom

$equivalentClasses(< ClassExpression >, ..., < ClassExpression >)$

C.1.1.3 Disjoint Classes Axioms

$disjointClasses(< ClassExpression >, ..., < ClassExpression >)$

C.1.2 Object Property Axioms

C.1.2.1 Object Sub-properties

$subObjectPropertyOf(< subObjProperty >, < superObjProperty >)$

⟨*subObjProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩

⟨*superObjProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩

C.1.2.2 Equivalent Object Properties

$equivalentObjectProperties(< ObjectPropertyExpression >, ..., < ObjectPropertyExpression >)$

C.1.2.3 Disjoint Object Properties

$disjointObjectProperties(< ObjectPropertyExpression >, ..., < ObjectPropertyExpression >)$

C.1.2.4 Object Property Domain

$objectPropertyDomain(< ObjectPropertyExpression >, < ClassExpression >)$

C.1.2.5 Reflexive Object Properties

$reflexiveObjectProperty(< ObjectPropertyExpression >)$

C.1.2.6 Transitive Object Properties

$transitiveObjectProperty(< ObjectPropertyExpression >)$

C.1.3 Assertions

C.1.3.1 Class Assertions

$classAssertion(< ClassExpression >, < Individual >)$

C.1.3.2 Positive Object Property Assertions

$objectPropertyAssertion($
$< ObjectPropertyExpression >, < sourceIndividual >, < targetIndividual >)$

⟨*sourceIndividual*⟩ ::= ⟨*Individual*⟩

⟨*targetIndividual*⟩ ::= ⟨*Individual*⟩

C.2 Class Expressions

⟨*ClassExpression*⟩ ::= ⟨*Class*⟩
| ⟨*ObjectIntersectionOf*⟩
| ⟨*ObjectSomeValuesFrom*⟩
| ⟨*ObjectHasValue*⟩
| ⟨*DataHasValue*⟩

C.2.1 Propositional Connectives and Enumeration of Individuals

C.2.1.1 Intersection of Class Expressions

⟨*ObjectIntersectionOf*⟩ ::= objectIntersectionOf(⟨*ClassExpression*⟩ , ⟨*ClassExpression*⟩)

C.2.2 Object Property Restrictions

C.2.2.1 Existential quantification

⟨*ObjectSomeValuesFrom*⟩ ::= objectSomeValuesFrom(⟨*ObjectPropertyExpression*⟩ , ⟨*ClassExpression*⟩)

C.2.2.2 Individual Value Restriction

⟨*ObjectHasValue*⟩ ::= objectHasValue(⟨*ObjectPropertyExpression*⟩ , ⟨*Individual*⟩)

C.2.3 Data Property Restrictions

C.2.3.1 Literal Value Restriction

⟨*DataHasValue*⟩ ::= dataHasValue(⟨*DataPropertyExpression*⟩ , ⟨*Literal*⟩)

C.3 Property Expressions

C.3.1 Object Property Expressions

⟨*ObjectPropertyExpression*⟩ ::= ⟨*ObjectProperty*⟩

C.3.2 Data Property Expressions

⟨*DataPropertyExpression*⟩ ::= ⟨*DataProperty*⟩

C.4 Data Ranges

⟨*DataRange*⟩ ::= ⟨*Datatype*⟩

C.5 General Elements

Here, it is important to point out that some elements are enclosed by ' and ' characters. These characters are used to maintain the elements' syntax compatible with the XSB syntax. Most of the elements enclosed by these characters are strings that may contain special characters, such as the : character, or may have an initial upper-case letter. Since these elements appear in the axioms, they will be processed by our *reactive system*, meaning that they should have a compatible syntax with XSB.

For example, consider the following axiom:

```
1 subClassOf(Baby,Child)
```

where *Baby* and *Child* are classes. In this case, our reactive system would be assuming *Baby* and *Child* words as variables and not as classes, because both have an initial upper-case letter. To avoid that, both words most be enclosed by the ' and ' characters. The following axiom is written in the right way:

```
1 subClassOf('Baby','Child')
```

⟨*string*⟩ ::= a finite sequence of characters

⟨*languageTag*⟩ ::= a sequence of characters used to represent a language tag.

⟨*IRI*⟩ ::= ⟨*fullIRI*⟩
| ⟨*abbreviatedIRI*⟩

⟨*fullIRI*⟩ ::= an IRI used to represent a resource.

⟨*abbreviatedIRI*⟩ ::= a finite sequence of characters used to represent a resource name. In this case, the default ontology IRI will be used to identify the resource namespace.

⟨*Class*⟩ ::=' ⟨*IRI*⟩ '

⟨*Datatype*⟩ ::= ' ⟨*IRI*⟩ '

⟨*ObjectProperty*⟩ ::=' ⟨*IRI*⟩ '

⟨*DataProperty*⟩ ::= ' ⟨*IRI*⟩ '

⟨*Literal*⟩ ::= ' ⟨*typedLiteral*⟩ '
| ' ⟨*stringLiteralNoLanguage*⟩ '
| ' ⟨*stringLiteralWithLanguage*⟩ '

⟨*typedLiteral*⟩ ::= ⟨*lexicalForm*⟩ '^^' ⟨*Datatype*⟩

⟨*lexicalForm*⟩ ::= ⟨*string*⟩

⟨*stringLiteralNoLanguage*⟩ ::= ⟨*string*⟩

⟨*stringLiteralWithLanguage*⟩ ::= ⟨*string*⟩ @ ⟨*languageTag*⟩