ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

**Project number:** 258109  
**Project Acronym:** FastFix  
**Project Title:** Monitoring Control for Remote Software Maintenance  
**Call identifier:** FP7-ICT-2009-5  
**Instrument:** STREP  
**Thematic Priority:** Internet of Services, Software, Virtualisation  
**Start date of the project:** June 1, 2010  
**Duration:** 30 months

---

D4.3: Conceptual models of the event processor and pattern recognition

**Lead contractor:** S2 Grupo  
**Author(s):** Sergio Zamarripa, Maria del Carmen Calle  
**Submission date:** June 2011  
**Dissemination level:** Public

---

Project co-funded by the European Commission under the Seventh Framework Programme  
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: An ontology of failures will be described, based on the output of other deliverables and on consortium experience. Based on this structure, a mechanism will be defined to represent cause-effect relationships in a way that allows an easy adaptation to information sent by the context observer and user profiler. The goal is to define a model that will support the event processor in the task of identifying failure causes and anticipating and preventing failures. One of the main challenges will be to achieve a model that is independent of the system software and the developing environment. Pattern recognition will be used over the stream of events in order to identify symptoms of failure before they occur and to detect performance degradation trends. Main goal of this package is to analyze the most promising possibilities to achieve preventive maintenance.
List of Figures

3.1 Abstract Event Correlation architecture overview ................................ 13
4.1 Types of Context Events from D2.3 [21]. ........................................ 15
4.2 User Event Taxonomy from D2.3 [21] ........................................... 16
4.3 Application Event Taxonomy from D2.3 [21] ................................. 17
4.5 Overview of Conceptual Error Model ............................................ 18
4.4 Cause Taxonomy ................................................................. 18
4.6 FixAction and other fixing concepts ............................................ 19

5.1 High-level View of a Rule Engine ............................................... 22
5.2 Forward Chaining ................................................................. 22
5.3 A CEP system interfaced with a target system ............................ 24
5.4 FastFix Adapter solution approach .......................................... 26
5.5 Ontology model and Jena inference subsistem ............................ 27
5.6 Jena, OWL-Api and Sesame scalability ................................... 29
5.7 Custom operators and Globals Example .................................... 30
5.8 Custom operators and Globals Example 2 .................................. 31

7.1 FastFix Event Correlation Architecture .................................... 37
7.2 Event Correlation Service Public Interfaces ............................... 38
List of Tables

3.1 Modules and Requirements ........................................... 14
1 Introduction

After the collection of context information as result of monitoring, fault detection is one of the main requirements of the solution. Inferring what problem is present in an application, exactly in the precise moment that it happens is a critical task of FastFix rationale.

Furthermore, in most of cases, the identification of situations will not be trivial. In this kind of systems, the total amount of events can be huge, which difficults the identification of specific problems. Indeed, it will be necessary to use correlation mechanisms to draw conclusions about what is happening.

In this document, we describe the event correlation system, which will process events generated by the context observer. Knowledge about the software maintenance domain is a prerequisite for the event correlation system, since most of conclusions about error classification, their causes, and user actions associated to the problems will be based on the collected experience, error taxonomies, bug patterns, and solutions to these errors.

Major challenges of the system are failure detection, failure prevention and root-cause analysis, which are key elements for software maintenance. First, the platform needs a powerful mechanism to be able to detect any issue, abnormal situation or performance degradation in order to notify the maintenance team about the problem. In addition to that, in some situations, event correlation will have the appropriate information to prevent the failure before it happens, taking the actions to avoid or mitigate its effects. Finally, the analysis of the cause or causes that lead to the current situation of undesired behavior is also a key factor that will help the users of FastFix to improve its efficiency.

Regarding the structure of this document, it has been defined as follows. Chapter 2 describes the main objectives and requirements associated to the event correlation system design, as well as non-functional requirements that should also be taken into account. Chapter 3 brings the components needed to achieve the requirements closer, as well as the modules to make them possible. Chapter 4 explains the conceptual model of the events that event correlation will work with. Chapter 5 describes the basics of the correlation engine, as well as ontology frameworks to interact with representation models. Chapter 6 explores how patterns of errors that are not yet known can be discovered by the system. Chapter 7 gives an overview of the conceptual model of the whole system and finally Chapter 8 presents some conclusions.
2 Event Correlation Requirements

2.1 Purpose

This chapter describes the functional and non-functional requirements of the event correlation system in FastFix.

2.2 Functional Requirements

1. **Failure detection.** This is the major FastFix requirement that event correlation can deal with. Context events coming from context observer will be processed inside the event correlation system, which must be able to correlate events coming from different sources of the monitored system (application, runtime environment, operating system,...) and firing a failure detection as result of that correlation. Time will also be important for this requirement, since it must be possible to detect the pattern of event sequences that lead to the failure (crash) or abnormal situation. The current issue must be classified based on the detected event sequence. In some cases, like SQL injection or input validation errors, it must be possible to recognize concrete expressions in textual user input. In addition to that, issues associated to usability will need to be detected for specific situations.

2. **Root-cause analysis.** Once an issue has been detected and correctly identified, the event correlation system has to identify possible causes that boiled down to the problem, since event correlation will have information about the previous state of the application, as well as the events associated to user actions, system resources, application performance that were gathered right before the failure occurred. In some cases also application configuration issues can be detected.

3. **Prevention and anticipation to already known faults.** Detection of a fault before it happens, communication with the context observer, and request for increasing the level of monitoring of certain sensors, in case that additional information can be gathered in the moment of the failure. Other advantages of the anticipation to the failure are, among others, the possibility to trigger the fault replication recording, as well as to be able to apply quick re-configuration or even a patch to escape from the problem.

4. **Patch selection and patch generation.** Interaction with server patch system to request for existing patches to be applied or to provide control objectives [14] as patches to be generated. Event correlation, in cases where the failure detection requires complex correlation, will be the first component to detect a failure, hence, it can be the first to trigger for an existing patch deployment or, in case that it is not available yet, request for patch generation based on a control objective. In case
that no patch is known for the current failure, a workaround might be available for it.

5. **Communication with Maintenance Engineering Support System.** First, in order to trigger the creation of a ticket, as a result of an issue detection. Second, with the aim to provide the developer or software maintainer some valuable information about the issue, once detected. Some of this information items are:

   a) Probable causes for symptoms of the current issue.
   
   b) Possible solutions for this issue (or this kind of issues, regarding its classification).
   
   c) Similar error reports associated with this issue (as a result of correlation) of several error reports.
   
   d) Retrieve patch or workaround for current issue from knowledge base
   
   e) Provide or recommend configuration fix for a configuration issue.

We have also investigated the possibility to apply machine learning techniques and frequent pattern mining to event correlation. Accordingly, it can also provide the following functionalities:

- Pattern recognition for unknown errors. Pattern recognition or pattern matching is based on the previous knowledge of the pattern to be found. Nevertheless, the event correlation system can benefit from the analysis of the events that cross the system, looking for frequent patterns of error, and providing new patterns based on machine learning techniques.

- Discover usage patterns. As well as the discovering of error patterns, it can evolve to the extraction of specific sequences of usage of the application being monitored, and as an evolution of that, the relationship between this sequence of user events and the final intention or task associated to the sequence.

- Discover relationships and statistics between other items: symptoms and probable causes, issue types and patches, issue types and workarounds or similar error reports.

- Provide a way to know which event/events fired a rule. In most cases, some rule-based systems deal with a lack a visibility of what actually happened when a rule was fired.

### 2.3 Non-functional Requirements

- **Versatility:** The model must be platform independent, in other words, it must derive conclusions independently of the monitored application and operating system of the FastFix client.

- **Efficiency:** The performance of the event correlation system must be acceptable from the software maintenance user point of view. Detection of issues, error cause identification and failure prevention should be available as soon as it is needed for maintenance purposes.
• Flexibility and re-usability: The event correlation component must be flexible and uncoupled enough from the other FastFix components to be used for other purposes in the future.
3 Event Correlation Logic

3.1 Introduction

Once the FastFix requirements to be covered by the event correlation component have been presented, we should describe how the event correlation system should realize the requirements 2.2.

3.2 Description of the logic

In this section we are going to explain which are the selected techniques that can provide the required functionalities, in an individual manner and in some cases, in collaboration. The lack of research in the integration of these technologies in a single system, especially between rule-based systems and description logics (ontologies), has required the development of prototypes of integration between these technologies, in order to ensure that they fullfilled and applied as solutions to the requirements accomplishment.

3.2.1 Rule-based Systems

From the different alternatives of event correlation techniques, as already evaluated in Deliverable 4.1: State of the Art of Event Correlation [25], rule-based systems allow to specify the behavior of the event correlation separating control and knowledge, so knowledge can be updated changing the rules, without changing the program code of the engine. As a second general feature, these rule-based systems allow to express domain (functional) operations in a way that is closer to natural language [26].

In general, rule-based systems are especially powerful to identify/detect situations, and to trigger the corresponding actions accordingly.

These systems allow detection of specific strings of characters that match a given regular expression, which would meet requirements related to specific textual user input (e.g. recognizing SQL injection). FastFix can also benefit from this feature to find specific exceptions in the trace log of the application, allowing to detect already classified errors.

In addition to that, rules are very useful and versatile to detect event sequence patterns, which is necessary to identify the sequence of steps that characterizes an issue. This is possible by specifying the sequence to be detected, directly in the rules.

Moreover, one of the major utilities of this kind of systems is the power to fire or trigger actions in terms of what has been detected. For example, once an issue has been detected in a rule, it can provide information to the maintenance support system about the new issue, creating a new ticket.
Combination of Ontologies and Rule-based Systems

We can benefit from the features of rule-based systems at a higher level, from a global perspective, focusing on detecting and identifying situations as well as indicating the actions to be taken when these situations occur. Moreover, the power of model representation, provided by ontologies, would allow to leave the details of such domain knowledge in the ontologies, simplifying the rules and making them more versatile.

Ontologies provide convenient ways to represent domain knowledge and allow reasoning about the elements in the domain, about relationships, equivalence or properties. In order to take advantage of the capabilities of ontologies (especially the reasoning capabilities), ontology reasoning can be incorporated in rule based systems, in order to provide the following utilities:

- **Categorization and classification.** Thinking on FR1 (fault detection), FastFix event correlation should detect issues from different types (e.g. performance degradation issue, input validation error, operating system error).

- **Concept relationships.** Moreover, not only the types of issue will be different, but also the actions to be taken depending on the type of issue. Ontologies can play an important role to allow for issue classification or categorization, and also to find the most suitable solution or action to be taken. Issue classification, as well as relationships about solutions, causes, symptoms can be represented in ontologies and they can be queried from the rule perspective. Consequently, the details of the issue taxonomy will be in the ontology, which hides the complexity of the rule, delegating these details to the ontology. The result is a query-able knowledge base storing relationships between causes, issues, symptoms, solutions and workarounds.

  This usage of ontologies give some alternatives to cope with all of the elements of requirement 5 (Communication with Maintenance Engineering Support System), which is based on providing information to the maintenance system about valuable information regarding the current issue. For example, it gives some ways to retrieve the patch/workaround that fits for the current issue from knowledge base, as well as a list of similar causes that could have lead to the occurrence of the issue.

- **Causality.** From the FastFix perspective, one of the most important relationships to be stored and reasoned about from the ontology is the causality relationships. Software maintenance domain knowledge about the type of events that lead to a failure or a performance degradation can be present in the ontology as relationships of type 'causedBy' or 'causes' between elements, as it will be shown on next section 4.2, in concrete in figure 4.5. Causality relationships are an important means for root-cause analysis FR3

- **Semantic event correlation.** Ongoing research on semantic event correlation is quite promising, since it allows to correlate a percentage of events that could not be performed with traditional syntactic correlation. It also provides a way to model knowledge with ontologies, and use it in combination with rules. Events coming from external sources contain information referring to concrete concepts, but depending on the information source the same concepts can be expressed with different words (the same semantic concept, but not the same syntactic word). This can be done with the use of an adapter that queries the ontology for semantically equivalent
D4.3 Conceptual models of the event processor and pattern recognition

3.2.3 Machine Learning

One of the main issues detected in the event correlation discipline while evaluating the state-of-the-art is the lack of automation to adapt to un-coded situations. In that sense, learning techniques can be useful in order to find patterns and shadowed relationships between events, as well as to discover new rules or maintaining the existing ones.

Machine learning for Event Correlation can definitely be applied to pattern recognition, in other words, use a machine learning algorithm to detect patterns in the data and represent the patterns in the form of rules. Discovered patterns, once validated by human experts (to avoid from false positive patterns), can be applied to an ontology update, rebinding the whole ontology with the new discovered patterns. Applying general pattern matching rules over the updated ontology, the event correlation system would be able to detect those learned patterns without changing the rules.

3.3 Event Correlation Modules

This section presents an overview of the modules that provide the required functionalities, and illustrates the communication between these modules. In Chapter 5 some of them will be analyzed and discussed. In Figure 3.1 four main modules can be distinguished:

- The adapter, which aims to prepare format events for correlation and add some important causality information to them.
- CEP Infrastructure, which uses the formatted events from adapter, will be especially powerful to identify/detect situations, and to trigger the corresponding actions accordingly in order to achieve preventive maintenance.
- Server Data Store, the component that includes the FastFix ontologies.
- Pattern Recognition Module, whose goal is the analysis of existing patterns to be discovered and update the ontology with new knowledge. This module will be further explained in chapter 6.

The modules that combine their technology with ontologies must be integrated with an OWL/RDF Api and/or an Ontology Reasoner. These modules are the Adapter and CEP Infrastructure, in order to implement requirements such as root-cause analysis, prevention and anticipation to already known faults and the communication to Maintenance Engineering Support System of probable causes for symptoms of the current issue.

An event coming from the context observer would start passing through the adapter, which checks, validate the event format and adds some fields associated with causality. In addition to that, the adapter is also able to perform required transformations in case that semantic correlation can be applied in the event correlation framework for the information
D4.3 Conceptual models of the event processor and pattern recognition

contained in the event. These transformations are associated with the ontology, for example, looking for concepts that can be expressed in a general way, allowing this information to be correlated with other semantically equivalent concepts. After performing required adaptations to the events, they reach the CEP Infrastructure, where correlation takes part in the process. This correlation can be combined with ontologies to be able to perform operations that need to interact with the representation model, like classification, inference of concept relationships and causality, that have been introduced in section 3.2.2. As result of correlation the system will build new complex events, like identification of symptoms, issues or causes. In other level, correlation also fires actions to be invoked to interact with other FastFix components [FR 5], once an issue is detected, a cause is identified or a possible failure can be prevented.

![Figure 3.1: Abstract Event Correlation architecture overview](image)

The pattern recognition module analyses the events coming from the context observer, as well as the complex events which result from correlation in several timestamps, providing ways to, not only provide statistics and aggregation data, but also to propose new patterns to be added to the ontology. Other feature that can be achieved in this module is the possibility to give information about which event/events fired a particular rule and which complex events were created as a result of that rule.

Table 3.1 shows what modules are involved mainly in what requirements:
### Table 3.1: Modules and Requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>ADAPTER</th>
<th>CEP INF. (in combination with Ontologies)</th>
<th>PATTERN RECOGNITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1: Failure Detection</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FR2: Root Cause Analysis</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FR3: Known faults prevention</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FR4: Patch selection</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FR5: Communication with Maintenance Module</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FR6: Pattern recognition for unknown errors</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FR7: Discover usage patterns</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FR8: Discover relationships and statistics</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FR9: Provide a way to know which event/events fired a rule</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NFR1: Versatility</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NFR2: Efficiency</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NFR3: Flexibility and re-usability</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4 Event Correlation Conceptual Model

In order to implement the requirements defined in Chapter 2, it is necessary to define some main concepts and the relations between them. So, in this Chapter, two conceptual model summaries are presented: first, “The Events” from D3.2 deliverable [16], because Events are the input data to correlation engine; and the other one from D2.3 [21] regarding Event Correlation Conceptual Model, where complex events are described, with concepts like Cause, Issue and Symptom and the relations between them.

4.1 Context Events

To apply Complex Event Processing (CEP) to a target enterprise system, we must be able to create events that denote activities that are happening in the system. There are two steps:

1. Observation step: First, we must be able to access and observe the activities at any level of a system. Observation must not change the system’s behavior.

2. Adaption step: Second, observations must be transformed into event objects that can be processed by CEP. Generally, this is done by tools called adapters.

The first step is one of the main goals of Context Observer. The Figure 4.1 and Figure show the observations or ContextEvent taxonomy described in D2.3 [21]:

![ContextEvent Diagram]

Figure 4.1: Types of Context Events from D2.3 [21].

ContextEvents can be distinguished into three main types:
D4.3 Conceptual models of the event processor and pattern recognition

- **UserEvent**, which denotes events originated from the user of the target application. Figure 4.2

- **ApplicationEvent** representing all events taking place within the target application. Figure 4.3.

- **EnvironmentEvent** constituting all events outside the target application. Figure 4.1

These events are atomics events, in other words, a record of an activity in a system. Using atomics events we can compound complex events.

![Diagram](image)

**Figure 4.2: User Event Taxonomy from D2.3 [21]**
A complex event is an aggregation of other events, called its members. The relationship between a complex event and its member is called aggregation. The member events of a complex event can signify activities that happen at different times and perhaps in widely separate components of a system. So a complex event can signify an activity that consists of several activities in different parts of a distributed system. Conceptually a complex event is in a higher level than the levels of its members [20].

Event pattern rules are used in CEP to create complex events. This rules are called aggregation rules.

**4.2 Conceptual Error Model**

As already described in D2.3 [21], some of the information needed in Event Correlation Requirements is called "conceptual error model". In this Section an overview of the three major concepts is presented, as well as some fixing related concepts.

The major concepts in Event Correlation Conceptual Model are Cause, Issue and Symptom.

Cause represents reasons for the appearance of certain Issues. There are different kinds of Causes: ImplementationCause and ExternalCause. The mayor type of ImplementationCause is Fault which is defined by [4] as "a design or coding mistake that may cause abnormal component behaviour". UserInteractionCause (abnormal or malign user behaviour) and EnvironmentCause (related to OS or Virtual Machine, JVM and Browser) are subtypes of ExternalCause. This Cause taxonomy is represented in Figure 4.4 in order to formalize it:
A *Cause* can give rise to one or several *Issues*. An *Issue* represents a certain *Problem*. Some common *Issues* are *Performance Degradation* or *Exception Thrown*. An important subtype of *Issue* is *Error* or *Erroneous State*. *Error* means that the system is in a state such that further processing by the system will lead to a *Failure*.

An *Issue* can manifest in several *Symptoms* and a specific *Symptom* can potentially be the manifestation of several *Issues*. *Symptoms* are all forms in which an *Issue* manifests and can be observed. A synonym of *Symptom* is *Failure* which is defined by [4] as “a deviation between the specification and the actual behaviour”.

An overview of these concepts are presented in Figure 4.5.

In order to remove *Causes* and mitigate *Symptoms*, which are some of the main tasks of FastFix, the *FixAction* concept must be described as an action carried out by maintenance engineer with the goal to address an *Issue*. Main subtypes of *FixAction* are:
D4.3 Conceptual models of the event processor and pattern recognition

- PreventiveAction, representing “a modification after delivery to detect and correct latent faults before they become operational faults” (definition taken from [17]).

- CorrectiveAction, representing “a reactive modification performed after delivery to correct discovered problems” (definition from [17]).

A FixAction applies a Patch, defined as “any modification to a source or object program” to the software system which can be a CodeChange, ConfigurationChange, or SupervisorModelChange. A Patch either remove a Cause completely (all Issues and Symptoms originating by that Cause will be remove too) from the application then it is called a Solution, or it removes or mitigates a Symptom only (e.g. by restarting the application). Then it is called a Workaround. Solutions are preferable to Workarounds but if the Cause is not known or it is time intensive to develop and apply a Solution, a Workaround is often used in practice. The relations between these fixing concepts, Cause and Symptom are shown in Figure 4.6.

Figure 4.6: FixAction and other fixing concepts
5 Event Processing

After summarizing event correlation techniques, we further describe and detail their main concepts and mechanisms. We will describe the CEP infrastructure, the adapter and the ontology related tools (OWL/RDF API and ontology reasoner).

According with Gruber definition [15] “An ontology is a specification of a conceptualization”, so ontologies will be used to represent the concepts and relations of the Event Correlation Model. In order to use this ontological information within the rules, and based on Bragaglia et al [3], in this section some tools and techniques will be described, and also how they will be integrated in order to achieve the combination of rules and ontologies.

These tools offer more benefits in terms of interoperability, extensibility and support. The first one is Drools, the forward chaining inference rule engine. The second one is the EC Adapter, whose main goal is to adapt the events from target system to a format that can be used by the rule engine. But not only a format that can be used by Drools, a format that can be used in an efficient way in order to generalize rules of Drools (allowing for semantic event correlation). The third, Jena, an open source Java-based framework for “semantic web” application, which will be used to manage ontological information, and the last one, Pellet, an OWL-DL Java-based reasoner which provides standard and advanced reasoning services for OWL ontologies, in order to exploit as much as possible the relation between concepts reflected in ontologies.

At the end, Custom Operators and Globals are described as the way to integrate reasoning tools within the rule engine.

5.1 Processing: Drools

As it’s said in Subsection 3.2.1, from the different alternatives of event correlation techniques, rule-based systems allow to specify the behavior of the event correlation separating control and knowledge and to express domain (functional) operations in a way that is closer to natural language. So, this section presents an overview of Drools focused in Drools Expert, the rule engine itself, and Drools Fusion, the Complex Event Processing module, which are the main Drools components in FastFix scope.

5.1.1 Description

Drools is a Business Logic integration Platform (BLiP) management system (BRMS) written in Java. It is an open source project (Apache License, Version 2.0) that is backed by JBoss and Red Hat, Inc.

From its beginning, Drools underwent many changes. The first version started with a brute force linear search. It was then rewritten in version 2.0, which was based on the Rete algorithm, which boosted Drools performance. In this version, rules were written mainly in XML, the next one (3.0) introduced a new .drl format. This is a specific
language specially crafted for writing rules. Version 4.0 of the rule engine had some major performance improvements together with the first release of a Business Rules Management System (BRMS). This formed the base for the next big release (5.0) where Drools became a BLiP. The platform consists of four main modules:

1. Drools Expert: The rule engine itself
2. Drools Fusion: Complex Event Processing (CEP) module.
5. Drools Solver: This is an optional module. It’s a search algorithm built on top of the Drools rule engine to solve planning problems (for example, creating timetables)

Another very important part of Drools is its Eclipse plugin. It greatly helps with writing and debugging rules and processes. It checks or syntax errors, offers auto completion, and has lots of other useful features.

Drools is an efficient and mature product and it has a very active and friendly community which is growing every year.

5.1.2 Functionality

Drools Expert and Drools Fusion are the most interesting modules in Fastfix scope. So in this subsection it will be described how is the rule engine (Drools Expert) and ReteOO algorithm and how event processing capabilities are adding into the platform (Drools Fusion)

Drools Expert

Drools Expert is the rule engine itself. Rule Engine is quite ambiguous in that it can be any system that uses rules, in any form, that can be applied to data to produce outcomes.

JBoss jBPM uses expressions and delegates in its Decision nodes which control the transitions in a Workflow. At each node it evaluates there is a rule set that dictates the transition to undertake, and so this is also a Rule Engine. While a Production Rule System is a kind of Rule Engine and also an Expert System, the validation and expression evaluation Rule Engines mentioned previously are not Expert Systems [11].

A Production Rule System is Turing complete, with a focus on knowledge representation to express propositional and first order logic in a concise, non-ambiguous and declarative manner. The brain of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules - also called Productions or just Rules - to infer conclusions which result in actions. A Production Rule is a two-part structure using First Order Logic for reasoning over knowledge representation.

The process of matching the new or existing facts against Production Rules is called Pattern Matching, which is performed by the Inference Engine. There are a number of algorithms used for Pattern Matching by Inference Engines, Drools implements and extends Rete, which is called ReteOO.
D4.3 Conceptual models of the event processor and pattern recognition

The Rules are stored in the Production Memory and the facts that the Inference Engine matches against are kept in the Working Memory. Facts are asserted into the Working Memory where they may then be modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion; these rules are said to be in conflict. The Agenda manages the execution order of these conflicting rules using a Conflict Resolution strategy.

There are two methods of execution for a rule system: Forward Chaining and Backward Chaining; systems that implement both are called Hybrid Rule Systems. Understanding these two modes of operation is the key to understanding why a Production Rule System is different and how to get the best from it. Forward chaining is "data-driven" and thus reactionary, with facts being asserted into working memory, which results in one or more rules being concurrently true and scheduled for execution by the Agenda. In short, we start with a fact, it propagates and we end in a conclusion. It is described in Figure 5.2.

Backward chaining is "goal-driven", meaning that we start with a conclusion which the engine tries to satisfy. Drools is a forward chaining engine, however Drools community are working hard in order to pushing Drools forward as a Hybrid Engine [9].

**Drools Fusion**

Events are processed by computer systems since they were invented, and throughout the history, systems responsible for that were given different names and different
methodologies were employed [12] It wasn’t until the 90’s though, that a more focused work started on EDA (Event Driven Architecture) with a more formal definition on the requirements and goals for event processing. Old messaging systems started to change to address such requirements and new systems started to be developed with the single purpose of event processing. Two trends were born under the names of Event Stream Processing and Complex Event Processing.

In the very beginnings, Event Stream Processing was focused on the capabilities of processing streams of events in (near) real time, where the main focus of Complex Event Processing was on the correlation and composition of atomic events into complex (compound) events. An important (maybe the most important) milestone was the publishing of the Dr. David Luckham’s book “The Power of Events” in 2002. In the book, Dr Luckham introduces the concept of Complex Event Processing and how it can be used to enhance systems that deal with events. Over the years, both trends converged to a common understanding and today these systems are all referred as CEP systems.

The current understanding of what Complex Event Processing is may be briefly described as the following quote from Drools documentation [12]

“Complex Event Processing, or CEP, is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes.”

Drools Fusion is the module responsible for adding event processing capabilities into the platform. Drools Fusion defined a set of goals to be achieved in order to support Complex Event Processing appropriately. This goals are based on the requirements not covered by Drools Expert itself, since in a unified platform, all features of one module are leveraged by the other modules. This way, Drools Fusion is born with enterprise grade features like Pattern Matching, that is paramount to a CEP product.

5.1.3 Rete Algorithm

Performance is an important requirement in most of the applications. To get the best out of any technology, it’s necessary to understand how it works. Better decisions can then made about how to use it, and what and where to optimize.

The Rete algorithm is an efficient pattern matching algorithm from implementing productions systems [1]. Pattern matching is the act of checking rules against known facts to determine which rules can be executed.

The advantage that this algorithm brings is efficiency; however, it comes at cost of higher memory usage, because it uses a lot of caching to avoid evaluation conditions multiple times. Rete generates a network from rule conditions. Each single rule condition is a node in the Rete network.

The Rete network is a rooted, acyclic, and directed graph, and it is represented by the KnowledgeBase class. The network is created when we add knowledge packages into the knowledge base. Rules are sequentially added to the network as nodes, and the network is updated as needed. The performance of the Rete algorithm is theoretically independent of the number of rules in the knowledge base.

The Drools Rete implementation is called ReteOO, signifying that Drools has an en-
enhanced and optimized implementation of the Rete algorithm for object oriented systems. The Rete algorithm works wonderfully in language systems where pertinent attributes about objects are directly asserted to the rules engine. In an object-oriented language, such as C++ or Java, an entire graph of objects can be reachable from a single named root object. Bob McWhirter of The Werken Company adapted Forgy’s original Rete algorithm to object-oriented constructs, creating the Rete-OO algorithm, adding some new types of nodes to the Rete network [10].

5.2 EC Adapter

Any implementation of CEP needs events in a particular format, and an adapter’s job is to put the events it gets into this format. So the role of adapters is to monitor the ContextEvents which comes from the target system, and convert them into events in formats used by CEP. If there are a variety of source events, it’s probably that a set of adapters will be need. However if the format event is homogeneous for all the source events, or there is just one source, CEP system will need only one adapter. Figure 5.3 represents this schema.

![Figure 5.3: A CEP system interfaced with a target system](image)

Adapters for CEP can be quite sophisticated. An adapter’s job it to transform events
D4.3 Conceptual models of the event processor and pattern recognition

from event sources format in to a data structure format expected by the CEP infrastructure, and deliver them to the appropriate components of the infrastructure.

Typically, an adapter carries out the following steps:

1. Listen for events in the target system’s communication layer
2. Translate events to a CEP internal format for event processing
3. Add causal attributes.
4. Output events to the appropriate components of the CEP infrastructure.

All adapters must do steps 1, 2 and 4, and although step 3 is optional, it will be very important in FastFix scope in order to implement causality requirements. “Add causal attributes” means prepare the events in order to construct a causality chain, by using event attributes to reference the cause of this event. For example, in Rapide event pattern language, which is a declarative computer language for writing patterns of events, the causality attribute refers to the events that had to happen in the system for this event to happen.

Step 2 will be key too, in order to make some semantic adaptations to generalize rules. For example, different Operating System will send different messages in order to communicate the same problem. If a common message is defined in the adaptor, the rule engine will need only one rule with the generic message, instead three rules, one for each OS message. This semantic function could be implemented using one adaptor for each OS, but with the use of ontologies, it will be implemented in just one adaptor, because the relationship between concrete messages and the generic one is established by the ontology. Just an OWL/RDF API will be necessary in order to obtain this relation.

In FastFix, events have an homogeneous format specified in D3.2 [16]. Each context event \( e \) includes information such as a time stamp, a duration and semantic context information describing the context event and its detected values as a set of RDF statements that instantiate a FastFix ontology.

So, Figure 5.4 represents the adapter solution approach for FastFix, where the Context Events are represented using an homogeneous format based on RDF, and although there will be several source events, the use of ontologies allows semantic adaptations in order to generalize rules and use just one adaptor.
5.3 Integration of Rules and Ontologies

In Subsection 3.2.2 we presented some combination of ontologies and rule-based systems benefits, such as simplifying the rules and making them more versatile because this integration would allow to leave the details of such domain knowledge in the ontologies. So, we need some OWL/RDF Api in order to use both technologies, ontologies and java, together.

Another reason for building an ontology-based application is to use a reasoner to derive additional truths about the concepts you are modelling. This task is performed by a reasoner.

So in this section we described Jena, an OWL/RDF Api, and Pellet, a reasoner.

Finally, Subsection 5.3.3 explains two technologies to integrate Jena and Pellet within the rule-based system: Custom Operators and Globals.

5.3.1 Jena

Jena is a semantic web framework for Java, which has been developed by the HP Labs until October 2009 [19]. Since then it is developed and supported by an open source community. Jena supports RDF, RDFS, OWL and SPARQL. Furthermore it includes the
D4.3 Conceptual models of the event processor and pattern recognition

possibility to read and write common RDF notations: RDF/XML, N3 and N-Triples. Furthermore the Jena project develops RDF servers, which provides an HTTP interface to RDF and supports SPARQL. Another important feature of Jena is the support of different reasoners, which infer additional knowledge.

**Accessing ontology files: Jena APIs** Many issues are related on how interfacing the different tools, and how to provide “triggering mechanisms” from one tool to other’s reasoning services. The former problem is accessing to the information in the ontologies. In order to achieve this goal, Jena framework and its extensions JenaBeans, provide RDF API and OWL API [19] to access ontology files and binding of concepts to Java classes:

- The RDF API contains basic methods for working with RDF graphs. The central interface is `Model` that is instantiated by the `ModelFactory`. While the `Model` provides methods for working with the RDF graph, the actual graph is represented by the `Graph interface`. The model provides methods like `createResource`, `createLiteral`, `createStatement` and `createProperty`. Additionally the resulting objects can be created, added and removed manually and provide methods itself for connecting for example a `Resource` with a `Property`. For investigating the `Model` it provides simple methods like `listProperties` or `listStatements` that list the containing data of the RDF graph. But the `Model` also provides advanced methods like `listSubjectsWithProperty` that help traversing the RDF graph.

- The central interface of the OWL API is the `OntModel` that derives from the `Model`. While the usual implementation of the `Model` uses a basic RDF graph as implementation of the `Graph interface`, the `OntModel` usually uses a `Reasoner` implementation, which then uses a RDF graph. For working with ontologies, the OWL API provides classes like `OntResource`, `OntClass` and `OntResource`. All classes provide methods for connecting them with other classes. For example, `OntClass` provides methods like `createIndividual`, for creating an instance of the `class`, or `subClass` and `equivalentClass` for inspecting the class.

**Inference subsystem: The reasoners** Gruber definition means that an ontology allows a programmer to specify, in an open, meaningful, way the concepts and relationships that collectively characterise some domain. One of the main reasons for building an ontology-based application is to use a reasoner to derive additional truths about the concepts you are modelling. A simple instance of this above could be: the assertion "ClassCastException is a Fault" entails the deduction "ClassCastException is an ImplementationCause". There are many different styles of automated reasoner, and very many different reasoning algorithms.

![Figure 5.5: Ontology model and Jena inference subsistem](image-url)
Jena includes support for a variety of reasoners through the inference API. Sometimes the usage of term *inference* and term *reasoner* is arbitrary, but henceforth the term *inference* will be used to refer to the "abstract process of deriving additional information" and the term *reasoner* to refer to "a specific code object that performs this task", according with Jena documentation.

A common feature of Jena reasoners is that they create a new RDF model which appears to contain the information that is derived from reasoning [8], as well as the information that were asserted in the base model (*Model*). This extended model (*OntModel*) is, nevertheless, still conforms to the contract for Jena models. So it can be used wherever a base model can be used. The ontology API exploits this feature: the convenience methods the ontology API provides can query an extended inference model in just the same way as a plain RDF model. Figure 5.5 shows one way to visualize it.

Graph is an internal Jena interface that supports the composition of sets of RDF triples. The asserted statements, which may have been read in from an ontology document, are held in the base graph. The reasoner, or inference engine, can use the contents of the base graph and the semantic rules of the language to show a more complete set of statements. This is also presented via a Graph interface, so the model works only with the outermost interface. This regularity allows us to very easily build ontology models with or without a reasoner. It also means that the base graph can be an in-memory store, a database-backed persistent store, or some other storage structure altogether (e.g. an LDAP directory) again without affecting the operation of the ontology model.

Included in the Jena distribution there are a number of predefined reasoners, such as RDFS reasoner, OWL reasoner and a Generic rule reasoner.

**Jena alternatives**

OWL Api and Sesame are some Jena alternatives. Seals project (Semantic Evaluation at Large Scale) describe in the paper *D10.3 Results of the first evaluation of ontology engineering tools* [13] how Jena obtains better results regarding conformance and interoperability than OWL-Api and Sesame when tools use them for processing ontologies.

Regarding scalability, in terms on time spent on importing and exporting the ontology, Sesame showed the best performance results in the scalability evaluation and could process large ontologies efficiently. Jena performed fast on processing small and medium sized ontologies (less than 10MB), while the large tests led to a significant execution time increase. OWL-Api has the worst scalability level, and it fails when load ontologies more than 1,3MB.

The Figure 5.6 shows scalability information related to Jena (JE), OWL-Api (OA) and Sesame (SE). The other columns correspond to ontology editors: NeOn Toolkit (NT), Protégé 4 (P4) and Protégé OWL (PO).
Although Jena and Sesame results are very similar, the reasoners supported by Sesame do not allow DL-style reasoning tasks like validation and subsumption checking, and Sesame currently does not support any of the interfaces (OWL API, DIG) that Pellet offers, so a direct link is currently not possible [6].

**Jena reasoners alternatives: Pellet**

Although some reasoners are included in Jena distribution, for complete OWL DL reasoning we must use an external DL reasoner such as Pellet, RacerPro or FaCT++. The Jena DIG Interface makes it easy to connect to any reasoner that supports the DIG standard. Performance (especially memory use) of the fuller reasoner configuration still leaves something to be desired [8]. Another way to integrate them is using direct Pellet interface. This option is much more efficient because it doesn’t have the HTTP communication overhead and provides more inferences.

Pellet, Racer and Fact are good reasoners, however Fact and Racer have some disadvantages against Pellet. FaCT++ just supports string and integer data type [5] and it doesn’t handle individuals [23] and RacerPRO is not open source and does not support nominals [18]. Pellet is a complete and capable OWL-DL reasoner with a good performance, extensive middleware, and a number of unique features. Pellet is written in Java and is open source under a very liberal license. It is used in a number of projects, from pure research to industrial settings [23].
5.3.2 Integrating Jena and Pellet within Drools: Custom operators and Globals

Once all the ontological information is generated by Pellet inference and it’s accessible using Jena OWL-Api, it will be used within the rule conditions using Custom Operators and Globals.

Although Drools present some predefined operators such as relational operators (<, >, >=), temporal operators (after, during, finishes) or other such as matches, which perform regular expression matching [1], it supports the creation of new operators in order to enhance rule conditions expressiveness.

Globals Drools variables are variables assigned to a session [1]. They can be used for various reasons as follows:

- For input parameters (for example, constant values that can be customized from session to session)
- For output parameters (for example: reporting, a rule could write some message to a global report variable)
- Entry points for services, which can be used within rules.

In FastFix scope Globals aim to provide information that is not available in the WorkingMemory to rules. They will use as source the ontology, using Jena and Pellet for this purpose. The obtained information will be processed by both Custom operators and/or predefined operators. In Fastfix Globals will provide necessary services/methods to use them within rules. For example the method “causedBy(Issue issue)” will be developed, and it will obtained related causes with this issue within the ontology.

An example of Custom operator isA and the method of the Global getRootCauseFromOntology is shown in Figure 5.7:

```java
rule "isCausedByFault"
    no-loop true
dialect "myel"
when
    $issue : Issue()
    $cause : Cause() from OwlGlobal.getRootCauseFromOntology($issue);
    Cause(idEntity == $cause.idEntity);
    Cause(this isA "Fault")
then
    fireIssueDetection($issue,$cause)
end
```

Figure 5.7: Custom operators and Globals Example

The rule isCausedByFaults obtains, for all the Issues in WorkingMemory, their known causes in the ontology, using the method isCausedBy of the Global OwlGlobal. Then, just the causes which exist in Working Memory are chosen (because it means that they have occurred), and between them, and using the Custom Operator isA the Fault causes are selected, because this Fault will be send to maintenance module.
Another example of both operators could be the showed in Figure 5.8:

```plaintext
rule "SQL Injection Drop Table"
   no-loop true
dialect "myel"
when
   $staticInformation: StaticInformationContext(this.app isA "SQLInjectionCandidate")
   exists UserInput (appId=$staticInformation.app.id, text matches ".*drop.*")
   $patch: Patch() from OwlGlobal.getPatch("SQLInjectionSolution")
then
   fireIssueDetection("SQL Injection: Drop table attempt");
   PatchContext.selectPatch($patch);
end
```

Figure 5.8: Custom operators and Globals Example 2

The rule “SQL Injection Drop Table” obtains information regarding the type of application from the StaticInformationContext, and just if the application is a SQLInjectionCandidate, the condition matches. Then if the UserInput for this application has keyed a text which contains “drop” string and if exists a patch related with SQLInjection in the ontology, all the rule matches and the message “Drop Table attempt” is sent to maintenance module, and the SQL Injection Solution patch is selected.
6 Pattern Recognition

6.1 Goals

As we have already introduced in section 3.2, machine learning techniques can be useful in order to find patterns and shadowed relationships between events, as well as to discover new rules or maintaining the existing ones.

Machine learning for Event Correlation can definitely be applied to pattern mining, in other words, use a machine learning algorithm to detect patterns in the data and represent the patterns in the form of rules. Discovered patterns, once validated by human experts (to avoid from false positive patterns), can be applied to an ontology update, rebinding the whole ontology with the new discovered patterns. Applying general pattern matching rules over the updated ontology, the event correlation system would be able to detect those learned patterns without changing the rules.

The idea of the pattern recognition module is to identify error patterns that are not yet available in the knowledge base.

In addition to the error pattern (ContextEvents⇒Issue), as we have advanced at the end of section 2.2, we can also discover the following pattern types:

- Symptoms⇒Issue
- Cause/s ⇒ Issue
- Issues ⇒ Patches
- Issues ⇒ Workarounds

6.2 Pattern Recognition Techniques

Pattern recognition starts in the working memory of the CEP Infrastructure shown in figure 3.1. This working memory is the space where events are inserted and correlated. These events can be either context events coming from context observer, as well as created complex events as result of aggregation of context events. Some examples of complex events are the ones shown in figure 4.5, in other words: symptoms, causes, issues derived from the correlation and aggregation of context events.

As observed in figure 3.1, an snapshot of events in the working memory in a period of time can be extracted with the use of listeners. Each snapshot of these events is periodically sent to the pattern recognition module, with the aim of identifying patterns that are repeated frequently or, with a different point of view, unfrequent patterns that can also be considered as valuable information.

As result of this pattern identification, several patterns will be proposed to a user, who will validate, based on software maintenance expertise, the accuracy and convenience of the proposed pattern, since it should be considered that, at least in a first stage, the
patterns can provide false positives, depending on the tuning of the minimum support (or threshold for a pattern to be considered as frequent). Hence, the type of machine learning applied at an early stage must be 'offline'. After tuning properly the system, the evolution to an 'online' (automated) pattern recognition can be further evaluated.

After validating the discovered pattern, the individuals in the ontology referring to concrete complex events in the pattern can be updated. Thus, the model will change and new relationships between the elements representing the events in the pattern will be inferred based on ontology reasoning. After that, a new inferred ontology will be available to interact with the events in general pattern matching rules. Therefore, the next time the new pattern events are present in the working memory, the pattern matching rule will be triggered, resulting in the detection of issues that were not known by the system based on expertise, and achieving to detect them based on machine learning observation.

In the following sections, we describe the most interesting algorithms and techniques suitable for the discovering of patterns (both frequent and unfrequent).

### 6.2.1 Frequent Pattern Mining

The frequent patterns to be recognized will be of two types, sequential and static. Static type is based on the detection of error patterns based on the occurrence of certain complex events of a concrete type (e.g. symptoms), they are frequently found in presence of other events (e.g. issues) and that might have a relationship (in this case causality). Sequential patterns, on the other hand, consist in detecting events that occur frequently in the same sequence, indicating a possible inherent relationship between them.

Depending on the type of pattern, there are specific algorithms for each type. The static type corresponds to the frequent-pattern mining algorithms, while the sequential type are detected by sequential pattern mining algorithms.

- **Frequent Pattern Growth Algorithm (FP Growth [2])**. This algorithm was created as an improved evolution of the Apriori algorithm, already mentioned in Deliverable 4.1 [25](3.4 Learning). These algorithms are based on iterative approaches known as level wise search. Throughout the level-wise generation of frequent item sets, it provides groups of item sets increasing its size, and evaluates the minimum support for each one of the items sets. An item set is considered as frequent it has at least the minimum support, previously specified. FP Growth [2] is focused on avoiding the bottleneck of the Apriori method, which is the candidate set generation, with the result of an order of magnitude faster that the Apriori algorithm. Among the open-source projects that provide FP Growth support, Apache Mahout 1 is a project that offers free implementations of scalable machine learning algorithms.

- **Frequent Sequential Patterns Algorithm (PrefixSpan [22])**. Sequential pattern mining allows the discovering of frequent subsequences as patterns, often used for customer purchase behavior analysis, web access patterns, natural disasters, DNA sequences, and so on. Specifically, PrefixSpan is a frequent sequential pattern algorithm which, given a set of sequences, and a minimum support threshold, it find all subsequences whose occurrence frequency in the set of sequences is no less than the minimum support. In addition to that, it has been proven in performance studies.

that it outperforms other sequential pattern algorithms, like GSP (Apriori-based) or FreeSpan.

6.2.2 Trace Analysis and Ontologies

Data mining techniques offer a means to automatically extract information from event traces. These techniques help identify frequent patterns in observed traces. In some situations however, information about unfrequent observations can also be extremely valuable. Determining patterns of an unfrequent trace can for instance help diagnose issues with very few occurrences. In case of program traces, such analysis is called Dynamic Analysis.

Automatic pattern detection on small data sets is a complex and often inadequate approach. Dynamic Analysis for instance usually requires human intervention. Although data collection from software programs can be performed in a systematic and automated way, its analysis involves human expertise. There is therefore a gap between the raw data collected from a system and the high level conclusion obtained by experts. Bridging this gap is an important challenge in the dynamic analysis and program comprehension community and is often tackle using visualization tools and techniques (e.g. [7]). As ontologies make it possible to represent high level knowledge and concepts but are still formal enough to be automatically treated by machines, they offer an interesting opportunity to help fill the gap between low level program traces and the outcome of their analyses.

One possible contribution to FastFix is to provide an approach and a tool in order to help the dynamic analyses perform on traces. This approach relies on ontologies where both concepts related to traces, such as context events, ordering and parameter values, as well as concepts related to their analyses, such as frequencies and invariants are encoded. Ontology frameworks usually come with a querying system that can itself rely on some reasoner. This makes it possible for experts to interact with the knowledge base consisting of system traces and their related concepts. This interaction is eased through the use of concepts together with some reasoning capabilities. For instance considering the ontology presented in Figure 4.5, query $Q_1$: "What are the traces containing an issue?" should return all the trace instances of the ontology that are identified as containing an issue, an error or an erroneous state. It is then possible to apply other queries to this set of traces, such as $Q_2$: "What is the most frequent method call event?". Therefore such an approach makes it possible for the expert to explore and reason about the traces collected from the system at a conceptual level, with the help of a reasoner, association between different concepts can automatically be performed and presented to the user, e.g. as error represents a subclass of issue, traces containing errors are also returned to the user whenever they request traces containing issues.

In FastFix, we propose to design and implement such an interacting tool. While requiring human expertise, this tool would also provide automatic help through its conceptualization and reasoning capabilities. This tool can be used to characterize traces, i.e. extract from specific traces the events that characterizes how they differ from other traces. This is of great interest as this can help diagnose issue, i.e. determine their causes, or at least characterize symptoms. In any case this approach helps deal with characterizing unfrequent traces as well as designing control objectives that can then be used by the self-healing component in order to generate patches.

There are however several challenges regarding such an approach. The first challenge
is to design general ontologies in order to represent concepts related to software events [May be this is discussed in one of the WP3 deliverable] as well as trace analyses. Another important challenge is concerned with the scalability of the approach. Although ontology framework such as Jena allow for persistent storage and querying that are not performed in memory, reasoners usually require at least part of the ontology to be loaded in memory. Very recently, non in-memory reasoning solutions have been proposed such as Trowl [24]. As this solution does not require in-memory reasoning, it offers an interesting means to reasoning on ontologies that possess a large amount of instances, such as program traces.
7 Event Correlation System Architecture

At the beginning of this deliverable Event Processing and Pattern Recognition requirements have been defined. Then we have analyzed and discussed some main concepts and the selected tools and techniques in order to implement these requirements. So, this last Chapter aims to present an overview of these tools and how they are integrated.

In Chapter 4, the main concepts related event correlation have been defined. These concepts and the relation between them are reflected in an ontology within the Server Data Store. These concepts are closely related with ContextEvents, specially with Symptoms.

From this relation between ContextEvent and concepts, that will be recognized by the Adapter, ContextEvents will be formatted before being inserted in the WorkingMemory in order to be correlated by Drools. This formatting process includes the addition of some causality attributes and some semantic adaptations, allowing an independent conceptual model from the system software and the developing environment, because the power of model representation, provided by ontologies, would allow delegate the details of such domain knowledge.

Once events are in WorkingMemory, they can be evaluated by the correlation engine within rules using Rete OOP Pattern Matching algorithm. Some important actions related with requirements such as failure detection, root-cause analysis and fault prevention are located in rules.

The expressiveness of rules condition is enhanced by CustomOperators, which have a double function: the implementation of functions that are not available in Drools predefined operators and the integration of ontologies and rules. In Section 5.3, another way to implement this integration is presented: the Globals, which in FastFix scope, aim to provide information that is not available in the WorkingMemory to rules. This information is the collected information within ontology.

Every component that use the ontology will use Jena, whose main characteristic is to provide an interface in order to access the ontology. In the case of Custom Operators and Jena, the use of Jena will be complemented by Pellet, a complete and efficient reasoner that allows infering more knowledge from the ontology.

Last, Pattern Recognition module, analized in chapter 6, it aims to detect event pattern that have been occurred in WorkingMemory, in order to, once they are validated, rebind the ontology and prevent the maximum number of failures.
In D2.3 [21] Event Correlation Service Public Interfaces is presented, the Figure 7.2 details it.

Event Correlation Service methods will use correlated and ontological information in order to, for example, provide the Causes or the RootCause of an Issue, or given a Cause, some other similar Causes or probable Solutions. In order to obtain this information the Inference Engine, Jena and Pellet are necessary.
D4.3 Conceptual models of the event processor and pattern recognition

Figure 7.2: Event Correlation Service Public Interfaces
8 Conclusions

In Chapter 2 requirements related to event correlation, such as failure detection, root-cause analysis and communication with Maintenance Engineering Support System, were analyzed in order to find then the most appropriate technologies and models for implementing them. Throughout the deliverable the need to combine and develop technologies to achieve these requirements can be appreciated. A clear example is how the expressiveness of rules conditions can be enhanced by Custom Operators, and how information which is handled within rules can be enhanced by the incorporation of knowledge that resides in ontologies to implement rules regarding requirements such as Failure detection and prevention.

The ontology integration within the correlate process has been key in order to achieve some main goals such as Concept relationships, Semantic Event Correlation and Causality, closely related with requirements like root-cause analysis, the communication of possible solutions for an Issue and to retrieve patch or workaround for current issue from knowledge base.

Last, pattern recognition module, whose most promising core foundations are FP Growth and PrefixSpan algorithms. These module aims to the ontologies are kept current, thus providing a better preventive maintenance.
Bibliography


[6] COMMUNITY, O. OpenRDF.


[10] DROOLS COMMUNITY DOCUMENTATION. Rete OO.


[15] GRUBER, T. What is an Ontology?

D4.3 Conceptual models of the event processor and pattern recognition


