**ICT PROJECT 258109**

Monitoring Control for Remote Software Maintenance

<table>
<thead>
<tr>
<th>Project number:</th>
<th>258109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Acronym:</td>
<td>FastFix</td>
</tr>
<tr>
<td>Project Title:</td>
<td>Monitoring Control for Remote Software Maintenance</td>
</tr>
<tr>
<td>Call identifier:</td>
<td>FP7-ICT-2009-5</td>
</tr>
<tr>
<td>Instrument:</td>
<td>STREP</td>
</tr>
<tr>
<td>Thematic Priority:</td>
<td>Internet of Services, Software, Virtualisation</td>
</tr>
<tr>
<td>Start date of the project:</td>
<td>June 1, 2010</td>
</tr>
<tr>
<td>Duration:</td>
<td>30 months</td>
</tr>
</tbody>
</table>

**D4.4: First iteration prototype of the Event Processor**

<table>
<thead>
<tr>
<th>Lead contractor:</th>
<th>S2 Grupo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s):</td>
<td>María del Carmen Calle, Sergio Zamarripa</td>
</tr>
<tr>
<td>Submission date:</td>
<td>February 2012</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>Restricted</td>
</tr>
</tbody>
</table>

Project co-funded by the European Commission under the Seventh Framework Programme

© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document reports on the implementation of the 1st prototype of the Event Correlation component, which focuses on the Event Processor, as well as its architecture and its main components. After the definition of the Event Correlation conceptual model, it is the next step to show the details of the architecture, design and implementation of the event processor, in order to accomplish the functional requirements of this component, which is the core of FastFix. Source code can be accessed from the FastFix repository¹.

¹https://repository.fastfixproject.eu/svn/fastfix
# Contents

1 Introduction 5

2 1st Prototype implementation of the Event Processor 7
   2.1 OSGi event correlation bundle 7
   2.2 Adapter 9
      2.2.1 Communication 10
   2.3 Rule engine 10
      2.3.1 Used libraries 10
      2.3.2 Firing rules 11
   2.4 Rules-Ontology Integration 13
   2.5 Sequence Diagram 15
   2.6 Report manager 20
   2.7 Event Processor Metrics 20

3 Issues and next steps 22
   3.1 Issues 22
   3.2 Summary and next steps 24

Bibliography 25
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Event Correlation Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>UserClick Detection rule</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of ReteOO graph</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>ReportManager: Example of error report</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Loading ontology</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Custom Operator isA</td>
<td>19</td>
</tr>
</tbody>
</table>
1 Introduction

The context of this document focuses on the FastFix project and specifically in the event correlation component, whose primary task is to detect error conditions and the acquisition of additional information that will be useful to the software maintenance team, through processing of large amount of information collected by the platform through the Context System. For that reason, FastFix requires an event processor to understand what is currently happening in the context of the application and what happens once detected, collecting more data to help determine the cause of such situation, through the detection of patterns of behavior associated with specific types of error, due to prior knowledge of their nature.

As background, this document is especially connected with the deliverable “D4.3e Conceptual model of the event processor and pattern recognition” [3]. This document gave a first overview of the foundation and conceptual model of the event processor, in charge of detecting issues and analyzing them, trying to extract possible causes and additional information that can be helpful for a software maintenance team. Additionally, it also explains the basis for pattern recognition, to be used over the stream of events in order to identify symptoms of failure before it occurs and to detect performance degradation trends. D4.3e specifies the functional requirements to be met by the component, such as issue detection, root-cause analysis, prevention and anticipation to already known faults, as well as other associated to information to be provided to the Maintenance Engineering Support System (like probable causes, possible solutions or similar bug reports).

Additionally, D4.3e sets out the main lines of the conceptual model, which defines the context events, which are input events to the system of correlation and complex events, which are the events resulting from the processing of multiple context events, storing higher level concepts either by aggregation, temporal relationship, and the information content of these events. These complex events are represented in the conceptual error model (described in Figure 4.5 of D4.3e), which can result in detected symptoms through the processing of multiple context events, which in turn generate one or more issues, which in turn can be related by certain causes (which can be confirmed by the occurrence of certain context events).

The whole process is based on this rationale applied in a rule-based event processing
environment, specifically the Drools Expert system, explained in more detail in Section 2 along with the rest of the architecture, the same way that the integration of rules and ontologies, which allow to classify and categorize types of error, a central feature for the correlation component that allows the generalization of rules, avoiding the design of a different rule for each specific error, and reusing a single rule for the type of error it belongs to. Hence, with only one rule the system can cope with any error of the same type whose details are concretely reflected in the maintenance ontology, which is meant to store the nature of any kind of issue of this type, the concrete symptoms associated to it, and the relationship with possible causes, and even solutions to be applied as a final step in the whole process.

An important consideration to be remarked is the scope of the first prototype. D4.3e describes both event processor and pattern recognition rationales. This first prototype is focused on the implementation of the event processor, and it does not include any implementation of the pattern recognition module, which will be demonstrated in the next prototype. Hence, the prototype that has been developed is focused on the processing of context events, through the use of a combination of a rule-based system, which is composed of a working memory, which is the space where the events will get to the system and a rule engine, which will fire concrete rules against those events, depending on the information they store.
2 1st Prototype implementation of the Event Processor

This chapter aims to give an overview of the structure, functionalities and architecture of the first prototype implementation of the Event Processor, as well as to differentiate between what was reused from existing libraries and what has been made from scratch in the prototype.

2.1 OSGi event correlation bundle

First of all, the event correlation prototype should follow the architecture specified in chapter 11 of deliverable D2.3e User Requirements and Conceptual Architecture (Extended version) [2]. This architecture is based on OSGi, so the interaction between other bundles should be done through import and export of other OSGi bundles.

Hence, the higher level structure for the event processor prototype is the event correlation bundle, which is part of the FastFix server project. This bundle is a specialization and integration of the original prototype included in the following SVN location: Software/unintegrated_components/EventCorrelator. The prototype existing in that location was not integrated, and the architecture was not OSGi. Additionally, some functionalities were not included, especially the ones that were dependant on other FastFix bundles, like the retrieval of context events (these events were simulated), or the generation of error reports. Hence, the event correlation bundle was built from the basis of the original prototype, structuring it as an OSGi bundle and including the following functionalities:

- Communication with the context system, receiving real context events through a context bus.
- Adaptation of the events to be inserted in the rule engine working memory.
- Integration with ontologies (globals and custom operators).
- Logging capabilities.
D4.4: First iteration prototype of the Event Processor

- Configuration of the bundle.

- Report Manager: Generation of error reports (interaction with the error reporting bundle).

Apart from these functionalities, the first rules have been designed, focusing on symptom identification, pattern detection and cause identification.

In order to provide the mentioned functionalities, these are the main modules that have been implemented (also represented in Figure 2.1):

- Adapter: It aims to format context events for correlation. This module is responsible for retrieving any context event coming from the communications infrastructure, checking its format and make any modification needed to take advantage of the information that it contains and finally, inserting the event in the working memory, getting ready for the execution of rules.

- Rule Engine. It uses the formatted events from the adapter. It will be especially powerful to identify/detect situations, and to trigger accordingly the corresponding actions in order to achieve preventive maintenance. This rule engine includes integration with ontologies, in order to reuse existing knowledge about diagnosis, causality and solutions for different types of errors, which is described in section 2.3. As we will detail in the next section, the main elements of this module are re-used from an existing open-source rule engine (Drools).

- Ontologies. The designed software maintenance ontology is stored in the server ontology, so any server bundle that requires the reasoning on the knowledge it represents can have direct access to it.

- Report manager. Depending on the type of error that is detected through the rules execution, the report will contain different relevant information. This report manager is responsible for formatting the information to be included in the error report and for communicating with the error reporting bundle for the creation of the report.

This chapter is focused in the design and implementation of these modules and how is possible the integration between them. Hence, we start the next section with the rule engine description.
2.2 Adapter

Context events coming from the FastFix client must be processed by the event correlation system. These events are generated based on the model specified in the common ontology that both context system and event correlation share. Nevertheless, a good practice is checking that every event is correctly formatted as the rule engine expects in its rules, in order to validate that everything is correct for the processing of these events. This practice ensures that, in case that there is an issue in any of the attributes of the context events, the system will be able to react conveniently, logging the formatting problem.

Apart from checking the correct format from the information in the common ontology, the adapter is responsible for slightly transforming the context events, so they can be processed by the rule engine and they can fire rules as they expect to.

This minimum set of attributes that should have an entity to be correlated, is defined in the class AbstractCepFact.java, and all the entities that will populate the working memory, in other words, all the facts (events to be inserted in the working memory), must inherit from it. Hence, the adapter gets the original context events and ensures the inheritance from AbstractCepFact, retrieving all the relevant information contained in the context event.

Apart from the context events, the rule engine will also process other type of events, the complex events. These events will be the result of processing the context events, inferring higher level concepts, such as identified symptoms, issues detected (through identification
of patterns of these symptoms), as well as identified causes. The insertion of these complex events in the working memory is done in the consequent (actions) of the rules, instead of the direct insertion of context events.

The different types of context events are detailed in section 4.1 of D4.3e [3], as well as the conceptual error model, where all possible types of complex events are described.

### 2.2.1 Communication

This prototype is using a ContextBus between the context system bundle and the event correlation bundle, in order to build the communication. Nevertheless, this is not the final solution to communicate both systems, since the communication infrastructure is currently being designed and developed.

Hence, taken this into account, we have used a context bus in order to inject real context events from the sensed information in the client context bundle. In the event correlation, we have developed a ContextListener, which is in charge of detecting any new injected event in the context bus, notifying it to the ContextListener. Each context event is extracted from the context bus, one by one. One of the things that should be improved in this prototype is the use of the FastFix communication infrastructure, getting more than one context event, which would result in a better performance of the whole system.

### 2.3 Rule engine

As a reminder of what is described in the conceptual model [3], rule-based systems are characterized by the use of rules, which serve to model the system behavior, in other words, to express the desired reaction of the system to events. This kind of systems provides a (close-to) real-time response, which is an important non-functional requirement, especially to ensure a quick reaction to provide information to software maintainers. Hence, the kind of correlation to be applied in FastFix should include time, since there are strong temporal relationships between the context events to be considered.

The core of the rule engine is based on open-source libraries that we describe in the following subsection.

#### 2.3.1 Used libraries

In order to implement the core of the rule engine, the event correlation system is based on Drools. Drools Expert and Drools Fusion are the most interesting Drools modules in
the Fastfix scope, since Drools Expert is the rule engine itself, and Drools Fusion is the Complex Event Processing (CEP) module.

Drools Expert is a rule engine and an expert system, in other words, a production rule system. The concept of rule engine is quite ambiguous in that it can be any system that uses rules, in any form, which 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. The brain of a production rules system is an inference engine that is able to scale to a large number of rules and facts (events in the working memory). The inference engine matches facts and data against production rules - also called productions or just rules - to infer conclusions which result in actions (in our case, the creation of an error report, by instance). A production rule is a two-part structure using First Order Logic for reasoning over knowledge representation.

Focusing on our concrete implementation of the prototype, we have build the rule engine using Drools Fusion with a real-time clock implementation, what is done setting the session clock to the value: ClockType.REALTIME_CLOCK. This value allows that it internally uses the system clock to determine the current timestamp. This configuration is performed in our implementation of the AbstractDroolsEngine, which is the TemporalDroolsEngineServiceImpl class.

Additionally, the event correlation bundle needs a mechanism to notify changes in the rules, while it is running. Hence, we have built a resource notifier, using the ResourceChangeListener provided by Drools. Hence, if the rule engine is started and a rule is changed, the changes are automatically applied. This mechanism can be useful if any of the mechanisms for pattern recognition require any change in the rules. Nevertheless, since the conceptual model is based on using the rules to represent the application logic (business of FastFix) (while the ontology represents the software maintenance domain knowledge), we will only need to make changes in the rules if there is a need to dynamically change the response of FastFix (apart from new discovered patterns, which will be updated in the ontology).

### 2.3.2 Firing rules

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 the Rete
algorithm, which is called ReteOO. 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.

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. These 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.

The detection of a concrete user interaction could be an example of pattern matching. In this case the Context Event gathered by the Context System is a UserClick, which is inserted in the Working Memory as a fact. At that moment, the rule 2.2 will be fired, because the UserClick fact has been matched against it. When this rule is fired the message “A user has clicked a button” will be sent to LoggerContext. The user can interact with the application in other ways, but the message will be sent only if a button is clicked and if the application associated to the context event is the specified in the rule.

```
rule UserClickDetection

when

 UserClick(applicationId="Moskitt")

then

 LoggerContext.logger("A user has clicked a button")
```

Figure 2.2: UserClickDetection rule

This is just an example of a rule, but it will help us on explaining the evaluation and activation lifecycle of a Drools rule. The whole process starts with the injection of an event of type “UserClick” in the working memory, as well as other events of different types. These types of events can be considered as objects, and the rules will be evaluated against these objects. After inserting the event in the working memory, the rule engine begins to evaluate the rule. In Drools, all rules are converted as a graph, which is generated by the ReteOO algorithm. The resulting graph will be something like the following figure 2.3:
In the resulting graph, each node of the tree will be one of the following types:

- **EntryPointNode**: Every event in the working memory is routed to this node.
- **ObjectTypeNode**: The system just keeps the events of the current event type (UserClick).
- **AlphaNode**: Node for matching constrains (applicationId=="Moskit")

Once all the conditions are satisfied, the graph will end in the following node:

- **TerminalNode**: If all conditions are satisfied, the rule consequent will be executed.

This is the basic lifecycle of the rules evaluation performed by the rule engine. Considering the complexity and number of rule conditions, the generated graph will also be more complex.

### 2.4 Rules-Ontology Integration

One of the main research lines of the event correlation system is the benefits of the combination of rule-based systems and ontologies. The main rationale for this combination is the separation of the FastFix application logic (rule) from the software maintenance knowledge (ontologies), which allows taking further profit, if we think about updating the ontologies with new recognized patterns of error. This solution allows being ready for the detection of these new patterns, since the ontology changes, but the rules do not, so the
insertion of these new patterns in the ontology allows being quickly ready to apply new knowledge acquisition.

Additionally, other benefit have been identified from this combination, like categorization and classification of different types of errors, representation of concept relationships, representation of causality and even semantic event correlation. These benefits were already introduced in section 3.2.2 of D4.3 [3].

Another reason for building this hybrid rule-ontology system is to use the ontology reasoner to derive additional information about the concepts we are detecting through the rules. This integration allows the rules design to focus on how we want FastFix to work and leave the details of reasoning over existing domain knowledge in the ontologies.

Regarding the concrete implementation of this integration, the prototype needs a semantic web framework to manage ontology interaction. In the case of this first prototype of the event processor, we have used Jena in order to interact with information in the ontologies. A project called SEALS (Semantic Evaluation at Large Scale) describe in the paper D10.3 Results of the First evaluation of ontology engineering tools [1] how Jena obtains better results regarding conformance and inter-operability than OWL-Api and Sesame when tools use them for processing ontologies.

Another reason for building an ontology-based application is to use a reasoner to derive additional information about the concepts we are modeling. This task is performed in the prototype by 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 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 [3]. However, Pellet provides another way to integrate them: use directly Pellet interface. This option is much more efficient because it doesn’t have the HTTP communication overhead and provides more inferences.

There is yet another consideration to take into account from our prototype, since we already have a way to work with information in the ontologies (Jena) and a reasoner (Pellet), but we still have to connect rule execution with ontology reasoning. We have created and OWLManager to centralize interaction with ontologies from the rule engine. We have found two different ways to connect ontologies and Drools, the most elegant are Drools custom operators, which provide a way to develop new functionality to be expressed in the rules. This functionality includes the reasoning over the ontology to extract relevant information required from the software maintenance knowledge. Drools, by default, already has some predefined operators such as relational operators (\(<,>,\geq\) ), temporal operators (\(after,\ during,\ finishes\) ) or other such as matches, which perform regular expression matching. For the interaction with ontologies, nevertheless, there are no
existing operators, so we have designed some custom operators to enhance rule conditions expressiveness and functionality. Among the developed custom operators, we have implemented the “isA” custom operator, which allows inferring if a concrete context event is a subclass of an existing class. This is extremely useful for identification of concrete types of context events, especially to identify concrete symptoms that are related to concrete sub-types of context events (user inputs, configuration changes, resource usage, and so on).

Apart from custom operators, Drools allow to provide information that is not available in the working memory to the rules. This is done through Globals, a concept introduced in section 5.3.2 of D4.3 [3]. In the FastFix scope, globals are used to interact with the ontology, using Jena and Pellet for this purpose. The resulting information is processed by both custom operators and/or predefined operators. In Fastfix, globals provide necessary services/methods to use them within rules. For example the OwlGlobal class provides services as getRootCause(Issue issue), which inform of the possible RootCause of an Issue.

Custom operators make rules simpler because they can obtain some details from ontology, so it’s not necessary to specify them within rules. Hence rules can be more general, thus improving their maintenance. For example, a rule can detect a text input too large. The number of characters that are considered too many could be obtained from the ontology and in the case that this number will vary the rule should not be modified, only the ontology. In order to create custom operators we have implemented the EvaluatorDefinition interface for each of them, and in this class the name of the new custom operator (e.g. isA) and how it works. For example, in order to implement the operator isA we have created the class IsAEvaluateDefinition, which implements EvaluatorDefinition. Thus, within the method evaluate of this class we have developed how the operator isA should behave. In this method the OwlManager.java class is called in order to use some Jena classes that allow us to access to the information of the ontology. The method evaluate always return a boolean value. In this case if an object belongs to a class (both object and class name are passed as parameter) the method will return true. In next section a sequence diagram shows how this custom operator works.

2.5 Sequence Diagram

The following sequence diagrams aim to explain more clearly the interaction between the components described in last section. We have considered in this section that some of the main procedures of the prototype should be detailed in order to make easier the understanding of the prototype’s code. Among the procedures in the prototype, we will
describe some of the main actions in order of execution, in other words, we will first explain how the ontologies available are loaded as soon as the component starts. In addition to this load, we will detail how all the events received through the ContextBus are loaded into the working memory and how the rules of the event correlation system are fired against the existing events in the working memory. Other important procedure to understand the interaction between rules and ontologies is the description of the custom operators, which can be included in the rules and which allow to interact with ontologies while firing the rules. In our case, we describe in this section one of the developed custom operators (isA), whose function is focused on validating if a concrete symptom or error is a subtype of a concrete type of other symptom of error.

These are the main classes associated to the previously described packages/modules:

- **Correlation Service**
  - eu.fastfix.server.event.correlation.IEventCorrelationService
- **Adapter:**
  - eu.fastfix.server.event.correlation.Adapter
- **Rule Engine**
  - eu.fastfix.server.event.correlation.engine.TemporalDroolsEngineServiceImpl
- **Custom Operators**
  - eu.fastfix.server.event.correlation.internal.operators.IsAEvaluatorDefinition
- **Ontologies**
  - eu.fastfix.server.event.correlation.internal.model.owl.manager.OwlManager
  - org.mindswap.pellet.owlapi.PelletReasonerFactory
  - com.hp.hpl.jena.rdf.model.ModelFactory
  - com.hp.hpl.jena.rdf.model.InfModel
- **Report Manager**
  - eu.fastfix.server.event.correlation.internal.report.manager.IReportManager

The first diagram, shown in Figure 2.5 represents how the ontologies are loaded. The Correlation Service calls `loadOWL` method from the class `OwlManager.java`. This method reads and validates the ontology using Jena and Pellet classes (`ModelFactory.java` and `InfModel.java` are Jena classes). The ontology is loaded during bundle activation. After that, the bundles is ready to receive new events through the ContextBus, calling the
Adapter to format the events into a CEP format that Drools can process. Finally, the events are inserted in the working memory and rules are fired.

The second diagram, shown in Figure 2.6, represents how the Custom Operator isA works. Once rules are fired and someone uses isA operator, the method evaluate from IsAEvaluateDefinition.java is called. Drools knows that it must look for this class when isA operator is invoked through the configuration of drools.packagebuilder.conf file. The evaluate method returns true if the object which invokes belongs to the class passed as a parameter. This information is in the ontology, so an OwlManager method is called: isSubClassOf(objectClass, className). This one calls Jena methods in order to obtain all the super classes of the object, and then return true if some of them are the class passed as parameter. This custom operator is useful to for categorization and classification purposes, as explained in D4.3e [3].

Report Manager is called within the execution of the rules, if the consequent of the rule associated to error detection is fired. The main method of the class is sendReport, which builds an error report with relevant information about symptoms gathered through the context events acquisition and the rules execution. One of the main functionalities is the creation of an error report conveniently adapted to the type of the detected error, the relevant information on the symptoms and the possible causes, as shown in figure 2.4, which is a real result of the event processor prototype.

![Figure 2.4: ReportManager: Example of error report](image)
Figure 2.5: Loading ontology
D4.4: First iteration prototype of the Event Processor

Figure 2.6: Custom Operator isA
2.6 Report manager

Once an error has been detected through the execution of rules, there is a need of a mechanism to send to corresponding error report, with the subject of the issue, as well as its description. This description should contain relevant information about the relevant events involved in the detection of the error. Required information in the report will be different depending on the type of error, as well as the format and content of the subject and description of the error report.

Apart from the information represented in the error, the mechanism should finally communicate with the error reporting bundle, sending the error report object to be created as a ticket in the bug tracking system.

The main consideration to keep in mind while implementing this first iteration has been the possibility to dynamically create error reports, with different information depending on different types of errors. This has been done through the implementation of the ReportManager, which takes relevant information from the consequent of the error detection rule. One of the main contents is the type of error, which is represented in the pattern (from the ontology). The type of error is already associated to the information that should be included in the report, hence, the report manager takes this information to build the error report conveniently, gathering information about the concrete attributes to be selected and represented from each one of the symptoms that compose the pattern.

2.7 Event Processor Metrics

The Event Correlator was implemented in Java using OSGi\(^1\) and the Eclipse IDE\(^2\). The code is available in the FastFix repository\(^3\). Table 2.1 shows some metrics obtained from the code implementing the event correlator. The metrics were obtained running the Sonar\(^4\) tool.

\(^1\)http://www.osgi.org/Main/HomePage
\(^2\)http://www.eclipse.org/
\(^3\)https://repository.fastfixproject.eu/svn/fastfix
\(^4\)http://code.google.com/javadevtools/codepro/doc/index.html
D4.4: First iteration prototype of the Event Processor

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>2966</td>
</tr>
<tr>
<td>Classes</td>
<td>62</td>
</tr>
<tr>
<td>Number of Packages</td>
<td>16</td>
</tr>
<tr>
<td>Number of Files</td>
<td>59</td>
</tr>
<tr>
<td>Comments</td>
<td>11.2%</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>352</td>
</tr>
<tr>
<td>Rules Compliance</td>
<td>78.4%</td>
</tr>
<tr>
<td>Complexity/Method</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 2.1: Event Correlation metrics
3 Issues and next steps

This final chapter aims to stress the existing issues from the first iteration of the event processor prototype, in order to take them as the main objectives to focus on for the next iteration of the prototype of the event processor. The consideration of these issues will enhance the second iteration, focusing on the identified weaknesses and relevant risks of the whole process.

3.1 Issues

Performance and management of ontologies

One of the main risks that we should stress is the performance of the combination between rules and ontologies. Currently, with the examples used in the prototype, the performance is good. Nevertheless, even though we have already used real context events, the amount of these events is not representative enough of the real amount of events that FastFix can need in a real environment. Hence, we should simulate the arrival of a realistic amount of events, in order to validate the response times and performance of the adapter, as well as the response of the combination of rule execution and reasoning over ontologies, which is one of the main possible bottlenecks of the event correlation performance.

Hence, in the next iteration of the prototype we have decided to include a simulation environment, in order to evaluate this performance to a greater extent. After analyzing the results of this evaluation, we will focus on tasks to enhance this performance, fixing the bottlenecks of the system and analyzing how the system evolves with these modifications.

Communication system

As already mentioned in section 2.2.1, one of the things that should be improved in this prototype is the use of the FastFix communication infrastructure, allowing to get more than one context event. Currently, we are using the communication capabilities between bundles of the OSGi architecture. Nevertheless, this has been used as a proof of concept,
and the next iteration should include the integration with the FastFix communication infrastructure, currently under development.

In addition to the communication, and although the draft format is already defined, the final structure of ContextEvents must be refined.

**Generalization of rules. Ontology representation**

This prototype already includes the first rules for symptom identification, error detection through pattern matching and a first step for cause identification rules. Results for the error detection rule have proven to give good functionality with a high level of generalization, in other words, the rule for error detection through pattern matching do not include any dependency on the type of error, leaving all the details to the ontological representation of the error pattern.

Nevertheless, this is not the same situation for the symptom identification rules, which are quite coupled to the type of symptom to be identified. Hence, we should work on an evolution of the symptom identification rules, in order to make them more general and to decouple them from any existing software maintenance knowledge. One of the main elements to focus to reach this objective is to improve the symptom representation in the ontology, which will result in more general rules, since the concrete attributes of each type of symptom will already be represented in the ontology.

Regarding the cause identification rules, even though it is a good start for this kind of rules, taking into account the complexity of the cause identification, we should refine them, looking for a better representation of the causality relationships in the software maintenance ontology.

**Pattern recognition**

The next prototype to be implemented is the pattern recognition module. As a reminder, the pattern recognition module aims to detect event pattern that have been occurred in working memory and are not yet available in the knowledge base, so they can be applied, once they are validated, 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.

Interoperability between the event processor and the pattern recognition operations will be possible by updating information about new discovered patterns as part of the existing
ontology. Hence, lessons learned during the implementation of the first prototype of the pattern recognition module should be taken into account for the implementation of the second iteration of the event processor prototype.

3.2 Summary and next steps

This first iteration prototype has met the main objectives it has been created for, taking also into special consideration the compatibility with the pattern recognition prototype. Hence, the road is already prepared for the implementation of the pattern recognition module. Additionally, it is useful for identifying the main tasks to focus on for the second iteration, based on the identified issues, the main risks associated with the event correlation operation and other components it interacts with.

Next steps to take within this prototype will be issue oriented, with special consideration on performance evaluation, as well as improving the rules in order to apply to a broader set of scenarios and enhancing the representation of symptoms, error patterns and cause relationships in the software maintenance ontology.
Bibliography

