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

D4.2: Domain Specific Language for Event Correlation

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

Project co-funded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: With the purpose that correlation rules can be easily expressed in the domain of the FastFix Project, we first examine the steps to be followed to create a Domain Specific Language, making rules for software execution monitoring easy to define. After defining this DSL creation process, we define the DSL itself, with the challenge of making it understandable for both application developers and machines and generic enough to be application, system and runtime environment independent.
## Contents

1 Introduction 6
- 1.1 DSL definition 6
  - 1.1.1 Need of a DSL definition 7
- 1.2 Domain: Software monitoring 7
  - 1.2.1 Fault detection 9
  - 1.2.2 Fault prevention 10
  - 1.2.3 Root-cause analysis 11
  - 1.2.4 Pattern matching 11

2 Design guidelines of a DSL definition 13
- 2.1 Language purposes 13
  - 2.1.1 General purposes 13
  - 2.1.2 Specific purposes 13
- 2.2 Guidelines 14
- 2.3 Core-requirements 16

3 DSL description 18
- 3.1 Levels of definition 18
  - 3.1.1 Modeling domain knowledge using ontologies 18
  - 3.1.2 Modeling domain logic using sentence mapping 20
- 3.2 Syntax, entities, operators and primitives 20
- 3.3 DSL specification 24
- 3.4 Conclusion 29

Bibliography 31
List of Figures

1.1 Fault detection: Classes, Individuals and Object Properties . . . . . . . . 10
1.2 Fault prevention: Classes, Individuals and Object Properties . . . . . . . 11
3.1 Causality chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
List of Tables

3.2 Example of DSL file for the security domain ........................................ 20
3.3 Temporal operators ............................................................................ 22
3.4 Primitives description ........................................................................ 23
3.5 Event attributes .................................................................................. 24
3.7 General DSL antecedent sentences for the FastFix domain ................. 25
3.9 General DSL consequent sentences for the FastFix domain ................ 25
3.10 Event DSL antecedent sentences for the FastFix domain ................... 26
3.12 Concrete DSL consequent sentences for the FastFix domain ............. 26
3.14 DSL antecedent structures for FastFix causality support ................... 27
3.16 DSL consequent structures for FastFix causality support ................. 27
3.17 Modifiers .......................................................................................... 28
3.19 Connections with other FastFix components .................................... 29
3.21 DSL rule example ............................................................................. 29
1 Introduction

This document is positioned in the context of the FastFix project, whose main goal is to provide a set of tools able to monitor on-line customer environments, collecting information about application execution and user interaction to identify symptoms of execution errors. Mainly focusing on two of the elements of the FastFix architecture, the Context Observer and the Event Correlator, the information monitored by the first of them, coming from the machine where the application is running, will be sent to the support center, via the communication infrastructure.

The Event Correlation module, with the aim of processing the events coming from the Context Observer, must deal with these events, which were generated during monitoring program execution and user interaction. This module must define the proper rules to be able to identify fault situations and determine the root-cause of these faults when possible. The mentioned rules must be able to represent cause-effect relationships between different entities present during this process, as well as the actions to be done by the application, once the situation to be identified has been matched by the correlation system.

1.1 DSL definition

Chiefly, we must define the concept of Domain Specific Language, first, to better understand what it exactly is and which role it can play along the FastFix rationale and second, to benefit its usefulness as a tool to represent rules and make them understandable and easy to create and modify, as well as a correct definition and usability of the DSL itself.

Instead of focusing on defining the best language for representing any kind of computing problem or situation, DSLs become particularly good for expressing a specific class of concepts, because of the particularities of the domain whose concepts and relationships are to be expressed.

Kolovos et al. claim [7] that a domain-specific language (DSL), whether used for model-driven development or programming, is a piece of critical infrastructure that is developed during the system engineering process. As such, a DSL has its own lifecycle, which in turn may encapsulate the lifecycles of many other system development projects. Understanding the requirements for DSLs in general, and in specific project contexts, is critical in order to improve DSL quality and ensure a direct correspondence between the requirements for system engineering projects and the functionality provided by the language. The quality attributes of a DSL — and its supporting environment (e.g., virtual machines, debuggers, integrated environments) — will have an impact on the quality attributes of the overall systems development process, and the resulting products. [7]

Probability the most interesting definition of DSL is the one provided by Walid Taha[13], who included in its definition the guidelines or characteristics that it should have, as follows:

- The domain is well-defined and central
1.1.1 Need of a DSL definition

Domain Specific Languages can serve as a layer of separation between rule code and the technical staff in charge of creating and maintaining the rules. The translation of this code into a language that is closer to the user's native language is essential, especially in cases where the size and complexity of the rule becomes high. If we look for an expression that accurately defines the need of a DSL, it should be “in order to hide complexity”. In the FastFix context, it is of particular importance, since most of rules’ complexity can be high while performing tasks to identify complex patterns, prevent future faults or represent and infer causality relationships.

1.2 Domain: Software monitoring

One of the most important things to keep in mind while defining a Domain Specific Language is the nature of the main entities related to software maintenance (domain objects). As indicated in deliverable D2.1 [12], there are many fault taxonomies in the computer science area, but a common general taxonomy has not been found yet. That common taxonomy would be helpful to describe at a high level the entities involved in software maintenance. Therefore, we propose the following entities in order to approach a definition of the domain objects:

- Measurement Sensor: Data collected by the Context Observer. Typically, the set of collected data will consist of the OS or VM, the Java Virtual Machine, the application itself and the browser (in case of a web application). About the kind of events collected by the Context Observer, we can find more detailed information in the document “D3.2: Conceptual model for context observation and user profiling”[4], where some concrete data examples are mentioned depending on the kind of information: user action, application reaction and execution environment (section 3.2 of the mentioned document).

At this point we recall the concept of “Context Atom” introduced in D3.1 [5]: “We use the term context atom in order to designate a specific aspect of context to be modeled. Context atoms, can be represented with five basic attributes: type, value, time stamp, source and confidence” These five attributes should be present in the MeasurementSensor entity.

On the other hand, conceptually and from the point of view of the ontology to be defined for event correlation, it would be related to objective measures or information on established patterns, symptoms, or directly with faults that previously had no symptoms. For example, the CPU usage percentage. We could make 2 limits \( l_1 \) and \( l_2 \), where \( l_1 < l_2 \):

- If \( \%CPU < l_1 \), then it means there is no abnormal situation.
• If $l_1 < \%CPU < l_2$, then it should be monitored to ensure that it continues to grow without good cause.
• If $l_2 < \%CPU$, then it could be an indicative that of a symptom, associated to a fault that may or may be not have shown symptoms earlier.

Other examples might be the connection pool settings of the application, the lack of free connections in the pool at any given time, and the fact that the log shows that the application failed because it has been too long while trying to connect.

• Configuration Information: Configuration data gathered on the monitoring environment, supplied by the Context Observer through a sensor. Recalling the previous example, the “connection pool settings” measurement sensor would be related to information on the monitoring environment settings, specifically the monitored application.

• Symptom: Fact that has occurred in the monitoring environment, which can serve as an indicator that something is not in a proper condition, and which is also provided by the Context Elicitation component through a sensor. The fact that some database connections are not released when the application is not performing any task with the database could be an example of a symptom that something may be wrong.

• Fault: A fact that has occurred in the monitoring environment, or expected to happen, that reveals a malfunctioning or unsatisfactory feature. The faults may or may not have associated symptoms.

  • In case that the fault has no associated symptoms, the data would be directly coming from the Context Observer, specifically from a measurement sensor. The reason why a fault may not have symptoms could be either because they were not actually raised, or because their relationship with the fault has not been recognized yet in our knowledge base. For example, the response timeout in a web application can be produced because there are no available connections in the pool. In case that the event correlator had no reference of a symptom representing the fact that the number of available connections was low, or it is not reflected in the ontology as a possible symptom of a fault. That would be an example of asymptomatic fault. Asymptomatic faults are also those produced by the application code at runtime, for example, a NullPointerException. If you enter a parameter with an empty value, and it operates without first checking its value, it will get a NullPointerException without previous warning symptoms.

  • Another possible situation can be the case that, even knowing the symptoms and the association with the fault in the knowledge base (ontology), the error occurs. Rather, if a set of symptoms is detected, and the association with a fault is known, a prevention measure can be thrown, in order to avoid the fault to happening.

• RootCause: Is the source event (or group of source events) responsible for triggering other possible causes leading to direct fault symptoms or directly to the faults if they are asymptomatic. The root causes shall be specified in the knowledge base
or ontology. In the example we have been using, the underlying cause would be the wrong configuration connection pool, as this would lead to poor management, leading to the fact that, at any given time, there were no free connections, hence reflecting in the error log that the application failed because it is taking too much time waiting for a response from the connection pool. The root cause is related to the symptoms, if they exist, or the failure, if absent.

- PreventiveMeasure: Given a set of symptoms some preventive measures can be applied in order to prevent a fault.

- Solution: It is the technical response associated to a root cause. It might be related with the patch generation module.

- Workaround: Given a failure or a symptom, if there is no associated solution to it, or it is not currently applicable, it is possible to execute an action whose aim is to mitigate the associated effects or symptoms, in other words, a workaround. For example, a possible workaround could be increasing the number of connections in the pool until the configuration data can be changed. The difference between workaround (or solution) and preventive measure, is a semantic difference: when the fault occurs, workarounds or solutions can be applied, when it doesn't occur, preventive measures are used.

- Patch: is the measure instance taken in order to fix or prevent a fault. Some examples could be:
  - A piece of code
  - Disabling modules
  - Configuration changes (OS, JVM, Application)

It is envisaged that, in order to simplify the model, the concepts: “configuration data”, “symptom” and “asymptomatic fault”, can be unified into the symptom concept, establishing the final subtype through its properties.

1.2.1 Fault detection

Applying the approach of the above entities in the event correlation ontology modelling, the extracted relations for fault detection are those shown in the figure 1.1:
So, it’s necessary that the domain specific language allows users to express different relations between Measurement Sensors, Symptoms and Faults, in order to achieve the Fault detection. Once the fault is detected, a solution or a workaround can be applied in order to fix or mitigate it.

A common but difficult example of fault detection is performance degradation. Performance problems that gradually occur might be difficult to diagnose. For example, if you detect a degradation in database server response time, it might not be obvious from looking at the current metrics which value is responsible for the slowdown. The performance degradation might also be sufficiently gradual that a change cannot be detected by observing the recent history of metric values.

1.2.2 Fault prevention

The domain specific language must allow to express relations between Measurement Sensors, Symptoms, Root Cause, Preventive Measure too, in order to implement Fault prevention rules. An example of these relations is shown in the figure 1.2.

Figure 1.1: Fault detection: Classes, Individuals and Object Properties
If there is an evidence that a fault can occur, increasing the level of monitoring to detect all possible symptoms and to apply the most appropriate preventive measure to avoid the fault.

### 1.2.3 Root-cause analysis

Root-cause analysis focuses on finding the trigger or initial reason of a fault and its symptoms. Given a certain fault, if there is no root-cause, or we don’t know which is the root-cause, an analysis must be performed for it to be identified.

For example, if the fault is an application error, the application log will contain the trace with the exceptions being thrown, leading to the “caused by” exception clause. Sometimes, this is the root cause, and in that case, the root-cause analysis would refer to the processing of the log traces during the time window of the fault occurrence.

Probably, in some cases, going through this causality chain is not that easy and we will be forced to rely on several rules to link one of the elements of the chain with other, until the system reaches the original. Hence, Root-Cause Analysis may be complex. Therefore, we need that the DSL is able to express this causal link between events.

### 1.2.4 Pattern matching

One of the most important features of any event correlation system, and one of the needs of FastFix is pattern matching, since the identification of event patterns is a key element when thinking on preventing failures.

As David Luckham claimed on “The Power of Events” [9], pattern language design is a complicated task because of two main issues:

- First, the resulting language elements must be easy to use, since event pattern languages can be oriented to users that are non-programmers.

- Second, there is a compromise between efficient pattern matching and power of expression.

If we try to identify which event pattern matching scenarios are needed in FastFix, in increasing order of complexity and difficulty of implementation, we would determine that these can be as follows:
- String pattern matching: This scenario corresponds to cases in which we need to identify simple patterns, including single event, and whose internal representation is a sequence of strings.

- Single-event, content-based matching: This is the selected scenario when we need to match single events, using content-based matching, in other words, the content of the event is used to decide the potential match.

- Multiple-event matching with context: This kind of situations is related to matching patterns of several events, using context information for the match, which requires more power of expression of the language and the representations of sets of events. The patterns will require logical operators (like: and, or) and guards with expressions. Some examples of this kind of pattern matching are the intrusion detection systems.

- CEP matching: This is the higher level of pattern matching. It requires representation of event executions with causal, aggregation and timing relationships. This requires a full set of relational operators.
2 Design guidelines of a DSL definition

2.1 Language purposes

2.1.1 General purposes

As stated by Karsai et al. [6], one of the first activities in language design is to identify the language uses in an early stage. This early identification allows keeping several considerations in mind while designing the language. For example, if the purpose is code generation the concepts and structures to be defined must be easy to connect one to each other.

In FastFix, the general purpose is double. First, it must allow rule code generation for identification of fault situations (symptom occurrence identification), as well as the actions to be executed by the application once these situations have been identified. And second, it must represent and document knowledge associated with the domain under study, in our case the software monitoring knowledge explained in previous chapter. This representation of knowledge must also be easy to understand by only reading the resulting rules in DSL format, even for a non-specialized professional.

2.1.2 Specific purposes

In order to narrow the specific purposes on Domain Specific Language we must identify the different types of sentences that FastFix will require to express. Since the sentences to be expressed will be used within a rule, we must first define the main elements of a rule. The rule antecedent is a set of logic sentences or clauses within a conditional logic rule that needs to be satisfied to match concrete situations, and rule consequent is a set of instructions that will be executed if the situation explained in the rule antecedent is matched. In this area, we should first split the kind of sentences we will need depending on the part of the rule where we are, in other words, the kind of sentences will be different in the antecedent than in the consequent of the rule.

Therefore, the rule antecedent in FastFix will be focused on identifying situations. FastFix event correlation component must be able to detect fault situations from the symptoms reported by the sensors, as well as to be capable of anticipating and prevent failures, hence able to identify that a failure is imminent. Thus, the language should be able to express when an event is occurring in a short period from the current observation, as well as being able to express a variant of that situation, pointing out an specific count of this type of event (e.g. \textit{the number of faults is greater than 1}). Depending on the purpose of the rule, either detect or anticipate a fault, FastFix will need to identify the occurrence of all or any of the symptoms of a fault. One of the situations we must also represent is the key concept of event correlation, the occurrence of any type of event
while another event happens, usually pairing them with a common property value, which will be the pair’s key.

The rule antecedent will also need to express the identification of a chain of causality, in other words, the fact that an event or fact is directly related to the previous appearance of another. Consequently, in order to define/represent the root-cause of a fault, which means that an event’s nature is such that it is not due to the occurrence of any other event, hence it is at the end of a causality chain. The antecedent is also prone to check for, not only events, but information about context, for example, configuration information, which are not events by themselves, since they don’t happen, but they just are configured like that, unless the event is reporting a change of the configuration. In this line, the checking of the current level of monitoring would also be an example, as well as the current version of the operating system, or some information about the application configuration.

On the other side, the rule consequent will need to express other kind of sentences. First, the report of the occurrence of an event not reported previously (either a fault or a symptom). The reporting of these events can be used in further rule firing to correlate with other events coming from the sensors. In addition to that, the event correlator must be able to notify to the Context Observer the need to increase or decrease the level of monitoring, depending on the situations identified in the antecedent of the rules in charge of that concrete task. With the same spirit, the Patch generation component will be notified with the aim of providing information about some kind of failures to be corrected, which will be identified as Control objectives, and Fault Replication will also be notified once a failure can be anticipated, in order to start gathering and recording information that can be useful to reproduce the failure.

One of the features FastFix will implement is the error reporting, which will show a list of the current error (or tickets) reported by the system, as well as some information regarding the fault, like an issue tracking system. Therefore, event correlation will also be in charge of correlating any information related with the fault, attaching it to the error reporting system. One of the sentences to be expressed in some rule consequents will be the creation of a new ticket/issue/bug in the error reporting system. In addition to that, several actions related with the ticket will be provided, probably taking the ticket ID as a key to identify the information to be updated. For example, the use of the knowledge base associated to software faults will allow the coding of a DSL sentence like “Add possible solution y to the ticket with ID x”, as well as additional information regarding the error ticket, like the addition of a root cause or symptoms that were actually detected by the event correlator, and which actually occurred.

2.2 Guidelines

These guidelines [6] describe techniques that are useful at different activities of the language development process, such as the early language activities development process (purpose), language implementation (realization), definition of the different elements of the language (content), readable or external representation (concrete syntax) and internal representation (abstract syntax).

- Identify language uses early: The language uses must be identified as soon as possible because their early identification has strong influence on the concepts the language
D4.2 Domain Specific Language for Event Correlation

will allow to offer. For example, if the language uses need an efficient execution, the
high-level concepts must be designed for optimized the tasks.

- Ask questions: Once the language uses have been identified it’s necessary to know
the profiles of the people who will develop, review and deploy the involved programs
and models, in order to embed these forms of languages uses into de overall software
development process.

- Make your language consistent: Each feature of the domain language should con-
tribute to the purpose of the DSL. As an illustrative example we consider a platform
independent modeling language. In this language, all features should be platform
independent as well.

- Compose existing languages where possible: In order to make the development of a
new language easier, existing languages can be reused. There are some sophisticated
techniques, however, embedding of an existing language into another language is the
most general form of language reuse. In any case, care must be exercised to avoid
confusion: similar constructs with different semantics should be avoided.

- Reuse existing language, type systems and notations domain: The design of type
systems is a hard task, therefore it’s more economical to use an existing one than
developing it. Furthermore, an unconventional type system would be hard for users
to adopt as well. Thus, reusing languages and existing type systems is recommended
when possible. In the same spirit, if composing existing languages is not possible,
taking the definition of a existing language as a starter to develop a new one is a
better idea than creating a language from scratch. And if we talk about syntax,
existing notations should be adopted as much as possible instead of inventing a new
concrete syntax for given concepts.

- Use only the necessary domain concepts: When designing a language, only those
domain concepts, that contribute to the tasks, need to be used. To ensure this,
it’s recommended to define some models early because they will be a good basis for
feedback from domain experts, in order to validate the language definition against
the domain.

- Keep it simple: Simplicity language content enhances the understandability of a
language, therefore, introducing a new language in a domain produces work (de-
veloping tools and adapting processes) and even such the language is successfullly
introduced, unnecessary complexity still minimizes the benefit the language should
have yielded.

- Avoid unnecessary generality: In order to not unnecessarily complicate the language
and hinder a quick and successful introduction of the DSL in the domain, techniques
such as generalizing or parameterizing concepts for future changes in the domain
should be avoided.

- Limit the number of language elements: If a language has several hundreds of
elements, this language is hard to understand. To limit the numbers of elements for
complex domains two approaches can be applied:
D4.2 Domain Specific Language for Event Correlation

- Design sublanguages which cover different aspects of the systems
- The use of libraries that contain more elaborated concepts based on the concepts of the basic language and that can be reused in other models.

- Avoid conceptual redundancy: In order to avoid redundancy problems, having several concepts at hand to describe the same fact will be avoided when possible.

- Avoid inefficient language elements: Elements which would lead to inefficient code should be avoided in order to generate efficient code.

- Make elements distinguishable: A key requirement to support understandability is the easily distinguishable representations of language elements. In graphical DSLs, model elements representations should exhibit enough syntactic differences to be easily distinguishable. In other kind of languages, such as textual languages, keywords are often used in order to separate kinds of elements. Anyway, models must be designed from a reader's point of view because they are much more often read than written.

- Enable modularity: Modularization is a key technique to tackle systems complexity because it leads to a flexible and understandable infrastructure. Therefore, if modularization is applied, the language should provide a means to decompose system into small pieces that can be separately defined by the language users.

- Use the same style everywhere: It is often necessary to use several languages to specify all aspects of a system. The user could obtain some kind of intuition for a new language due to is knowledge of other ones if the same look-and-fell is used for all sublanguages.

- Align abstract and concrete syntax: The abstract syntax and especially its structure should follow the concrete syntax to ease processing, transformations and also presentation of the model. In order to achieve it some advices such as construct different abstract notations for different concrete syntax, the use of subclassing to represent concrete elements that have similar meaning and the independence between abstract notation and the context, must be followed when possible.

Summarizing, we must try to identify the main task of the language as soon as possible in order to define the best possible DSL domain concepts, trying to represent, in an easy and understandable way, just the necessary elements in the domain, reusing whenever possible, other languages, sublanguages or just elements, ensuring that abstract and concrete syntax are aligned.

2.3 Core-requirements

The core requirements for a DSL are as follows: [7]

- Conformity: the language constructs must correspond to important domain concepts.

- Orthogonality: each construct in the language is used to represent exactly one distinct concept in the domain.
D4.2 Domain Specific Language for Event Correlation

- Supportability: it is feasible to provide DSL support via tools, for typical model and program management, e.g., creating, deleting, editing, debugging, transforming.

- Integrability: the language, and its tools, can be used in concert with other languages and tools with minimal effort. This is essential to integrate the DSL with other facilities used in the engineering process. An alternative requirement for DSLs is extensibility, i.e., that the DSL (and its tools) can be extended to support additional constructs and concepts. However, we suggest that integrability is to be preferred as a requirement for DSLs as it preserves semantic coherence of the DSL, as well as the desirable requirements of conformity and orthogonality.

- Longevity: the DSL should be used and useful for a non-trivial period of time in order to ensure tool support, and to make it possible to quantify to the DSL stakeholders the payoff obtained from using the DSL. There is, of course, an assumption with this requirement (and with the use of DSLs in general) that the domain under consideration persists for a sufficiently lengthy period of time to justify the cost of building a DSL and supporting tools.

- Simplicity: this is a generally desirable language requirement: a language should be as simple as possible in order to express the concepts of interest and to support its users and stakeholders in their preferred ways of working.

- Quality: the language shall provide general mechanisms for building quality systems. This may include (but is not limited to) language constructs for improving reliability (e.g., pre- and postconditions), security, safety, etc.

There are additional requirements that, while useful and desirable, need not be necessary for building all DSLs. These are as follows:

- Scalability: the language provides constructs to help manage large-scale descriptions. Of course, some DSLs will only be used to build small systems.

- Usability: this includes requirements such as space economy, accessibility, understandability – characteristics that are desirable, and which may be partly covered by the core requirements (e.g., simplicity can help promote understandability). Many of the above requirements are also applicable to general-purpose modelling and programming languages. However, as we mentioned earlier, their relative importance differs when comparing general-purpose languages and DSLs.
3 DSL description

3.1 Levels of definition

Description Logics [3] already provide languages and tools for representing upon a certain domain. These languages are one of the best options for knowledge representation, classification of new concepts and recognition of individuals to a given hierarchy. Hence, these are suitable for representing domain knowledge.

On the other side, rule-based systems [3], instead, are good to express domain logic (or application logic), in other words, to describe the functional operations that handle information exchange, operations to be done by the application, like communication with other components, insertion of new events in the working memory or selection and retrieval of existing events in the working memory.

Combination of RBS and DL reasoners Domain-based applications could greatly benefit from such integration, at both the levels of language expressiveness and at the level of algorithms and reasoning tools. Some previous research about this combination of techniques has already been done. For example, in “D4.1: State-of-the-art of event correlation and event processing”[15] the Semantic Event Correlation is described as an hybrid technique of model-based and rule-based correlation. Semantic technologies are usually associated with ontologies, because they have various advantages, such as making domain knowledge explicit, and sharing, using and reusing this information between people and software agents.

Hence, specification of rules in FastFix can be done in two levels, that we will define as follows:

3.1.1 Modeling domain knowledge using ontologies

Introduction to ontologies and Description Logic Ontologies have become a major tool to represent, express and formalize domain knowledge in the last years. An ontology is, following a popular definition of T. R. Gruber, “an explicit and formal specification of a conceptualization.” Knowledge about objects and relationships in a certain domain is formalized in a model and the model itself is expressed using an ontology, which provides certain constructs to express the knowledge.

Many different ontology languages exist, varying in their intended purpose, expressiveness and the inference support they provide. Prominent examples of ontology languages are the ones building on Description Logics with OWL [11]. Description Logics is a certain kind of logic that was designed to be expressive enough to be useful to describe knowledge about a domain while being restricted enough to exhibit desirable computability properties (this tradeoff is known as tradeoff between expressiveness and computability). In the field of Description Logics, the computability properties of different modeling constructs have been studied intensively (see [2] for more details) and OWL maps to Description Logics
in order to reuse this research effort and reasoning technology already implemented for Description Logics.

**Definition of a DSL based on OWL**  OWL can be used as a meta language to define a DSL. In this approach, an OWL ontology is defined containing concepts of the domain (called classes in OWL), relationships between those concepts (called properties in OWL) and constraints on the usage of concepts and relationships. This OWL ontology forms already a DSL and a DSL user can use and instantiate concepts in the ontology and assert relationships between the instantiated concepts. By doing so, the DSL user describes his domain of interest based on the vocabulary and constraints provided by the OWL ontology.

For example, if the OWL ontology defines concepts *Problem* and *Cause* and a relationship *isCauseFor* between *Cause* and *Problem*, a DSL user can assert that *DivByZeroException* is an instance of a *Cause*, *ApplicationCrash* is an instance of a *Problem* and *DivByZeroException* is one possible cause for *ApplicationCrash* (using *isCauseFor* property). A reasoner could then propose *DivByZeroException* as a possible cause when an instance of *ApplicationCrash* occurs. Further, the constraint that “every Problem has a Cause” can be asserted which implies that for every instance of *Problem* there must be at least one relationship *isCauseOf* from an instance of *Cause* to it (which may be asserted explicitly or not).

The approach to use OWL as a basis to define a DSL has been used by others, for example Martin et al. [10] describe web services or Kunze et al. [8] describe robots using a similar approach. Often, the inference methods of OWL are enriched by inference methods that are implemented by the DSL designers and that take relationships expressed in the DSL into account.

The advantage of using OWL as a basis for a DSL is that OWL is a standardized technology which eases interoperability and integration with other OWL ontologies and enables reuse of language constructs and reasoning support which comes with OWL. Another advantage described by Tobias Walter [14], who proposes a solution that allows the specification of DSLs with seamless integrated ontologies, consists in the suggestions and explanations to the DSL user by the OWL reasoning engine, because sometimes DSL users do not have complete knowledge about all domain concepts and they require more information of domain concepts to be used. Disadvantages are that OWL may not be pithy enough to articulate all knowledge that should be expressed or does not support all inferences that are desired. For example, OWL cannot express the relation *uncleOf* (*X, Y*) denoting that person *X* is the uncle of person *Y* (see Antoniou et al. [1] for more details on this example). In contrast, rules can express such a relation easily. In FastFix, we want to be able to express relations like *uncleOf* (*X, Y*) and hence we cannot use a pure OWL approach.

**Further work**  As mentioned above, FastFix can benefit from expressing information using ontologies but OWL is not expressive enough to model all information interesting for FastFix. Hence, we will continue to analyse and research how we can integrate OWL and rule based representations. We will report about the results of these efforts in subsequent deliverables, probably in D2.3 or D4.3.
3.1.2 Modeling domain logic using sentence mapping

The format of a DSL that converts a domain rule to an engine rule, fully understandable by a rules engine (e.g. Drools), looks something like a plain-text file. It is composed by several lines, and each of them, make sentence mapping possible. First, it is specified whether the sentence is suitable to be placed on the antecedent of a rule, as a condition ([when] or [condition]) or, in the other side, if it is defining an action, to be executed in the consequent of the rule ([then] or [consequence]). In Drools, comments start with the character: “#”. In addition to that, if we need to create conditions with a subset of constraints, we can express any of those constraints starting with the symbol: “-”. For example, a DSL example in a security rule-antecedent, to identify a malicious access to a Linux server would look like:

```
# Match against the then part of our security rules
[when] Any failed linux access to the server 4733
[when] -without previous failed access with same source, same destination
[when] -in a time window of 2 minutes, the count of failed accesses is 3
# The actions to be taken in the 'then' part
[then] Block external access to server 4733
```

Table 3.2: Example of DSL file for the security domain

We can think of DSL as a translator, since it defines how to translate sentences from the problem-specific terminology into conditions or actions present in rules.

As we will describe in section 3.3, in case that the final implementation includes the combination with ontologies, this sentence mapping also allows to express the high-level calls to the reasoner in charge of perform the domain knowledge “queries” to the DL reasoning system. Thus, the sentences to specify the integration with the knowledge base (or central information store), would be like: “any known fault associated to the current symptom list”, which would be expressing the connection at reasoner level.

3.2 Syntax, entities, operators and primitives

The sentences used within the language will consist of static words and placeholders (which will be identified by the property of its representation between brackets). The placeholder is a similar concept than the one described by David Luckham in “The Power of Events” [9], where it defines a concept for the variables in patterns, since they occupy the places that are “open” in a pattern. These placeholders will contain event types or entities, property values or constant values, depending on the expected kind of element. For example, in a sentence like “An event of type {Event} has a property with value {value} equal to {constant}”, we can identify several static words, as well as two placeholders, which can be replaced by any element representing either an event type ({Event}), a property value ({value}) or a constant ({constant}), once the sentence is used in a concrete rule.

Regarding the entities involved in the DSL, they will be based on the domain objects described in section 1.2. Hence, the following entities are needed in FastFix DSL:
D4.2 Domain Specific Language for Event Correlation

- Measurement Sensor
- Configuration information
- Symptom
- Fault
- Solution
- Workaround
- Patch
- Cause
- Root-cause
- Preventive Measure

These are the entities that can be identified at the present stage of the project, but it is worth to say that, as the project progresses, some other entities can be identified, either new independent entities, either some specialization of the ones already mentioned.

Once the main entities to be used in the DSL have been identified, other elements to express their relationships and temporal constraints are needed. First, regarding the root cause analysis mentioned in section 1.2.3, a mechanism to express causality relationships between the events in the cloud is a must. The implementation should allow the connection between the identifier of the event that represent the fault, and the identifiers of the list of the events of its causality chain. Hence, it is essential a common attribute, which we will call causality, referring to the events that “had to happen” in the system for this event to take place. A good approach is a vector of event identifiers, always keeping in mind that the interpretation of “had to happen” is system dependent and will be incorporated into the adapter for events in that system.

Logical operators are a good way to express meaning in the rules, which are always present in this kind of languages, since they are really useful to indicate wether a rule with a combination of two patterns must match both of them (and), either pattern or one pattern (or) and not the other (not).

One of the most relevant aspects of event correlation is time, since most of the conditions applied to filter events, to derive conclusions from several events or to identify situations have to take into account events timestamps or sequence. Hence, temporal relationships between events are to be expressed by means of temporal operators like the one defined in the following table:
### D4.2 Domain Specific Language for Event Correlation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>In case of pattern matching, it provides a way to match appearance of events with a specific timestamp. A good approach could be the use of a small range round the timestamp value that we can define with a concrete value, as a way to consider events as happening “at” an specific moment.</td>
</tr>
<tr>
<td>after-before</td>
<td>Both may apply to match complex patterns, describing the set of events, all of whose start times are greater (or less in case of before) than a concrete timestamp. They might be good partners for causality matches.</td>
</tr>
<tr>
<td>during</td>
<td>A way should be provided to specify whether events whose start timestamps are greater than T1, and whose end timestamps are less than T2.</td>
</tr>
<tr>
<td>over a time window</td>
<td>A sliding window is a way to scope the events of interest as the ones belonging to a time window that is constantly moving. The two most common sliding window implementations are time based windows and length based windows. For example, “over a time window of 30 seconds” will only pattern match the events that happened in the last 30 clocks ticks, discarding any other event older that that.</td>
</tr>
</tbody>
</table>

Table 3.3: Temporal operators

These operators can simplify writing conditions of the rules that refer to the timestamps and start and end times of events.

Finally, on a higher level of complexity, we can define sentences identifying more complex relationships that we will call *primitives*, as we describe in the following table:
### Table 3.4: Primitives description

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Syntax</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td><code>{EventA}</code>- <code>{EventB}</code>&lt;br&gt;Pair where key is <code>{keyName}</code></td>
<td>A place or part at which two things are joined (a property with the same value)</td>
<td>Example: Detected CPU overload and Virtual Memory high from the same process:&lt;br&gt;CPUOverload - VirtualMemoryHigh Pair where key is processId</td>
</tr>
<tr>
<td>Repetitions</td>
<td><code>{Event}</code> repetitions greater than <code>{threshold}</code></td>
<td>Condition to specify the occurrence of an event, on a specific amount of times</td>
<td>Example: Increase level of monitoring if number of faults is greater than 2:&lt;br&gt;Fault repetitions greater than 2</td>
</tr>
<tr>
<td>Group by (Demultiplexer)</td>
<td><code>{Event}</code> repetitions greater than <code>{threshold}</code> with the same <code>{keyname}</code></td>
<td>Combination of the pair and the repetitions primitive, allowing for defining appearance of repetitions of events with common properties</td>
<td>Example: Count of the faults in the same machine CPUOverload repetitions greater than 3 with the same idMachine</td>
</tr>
<tr>
<td>Followed by</td>
<td><code>{Event1}</code> is followed by <code>{Event2}</code></td>
<td>This primitive represents the causality between two events and its appearance into the working memory, with a subsequent timestamp. The combination of start timestamp and the causality properties of the events makes it possible.</td>
<td>Example: EventA-EventB sequence&lt;br&gt;EventA is followed by EventB</td>
</tr>
<tr>
<td>N Sequence</td>
<td><code>{Event1}</code> is followed by <code>{Event2}</code> AND <code>{Event2}</code> is followed by <code>{Event3}</code></td>
<td>Combination of the two-element sequence to provide a n-element sequence (with the use of the AND logical operator)</td>
<td>Example: EventA-EventB-EventC sequence&lt;br&gt;EventA is followed by EventB AND EventB is followed by EventC</td>
</tr>
</tbody>
</table>
3.3 DSL specification

First, entities declaration is important in this context, since it allows to define what kind of elements will be interacting within our language expressions. Moreover, the attributes of the most general entity, the event, must be established, to benefit from its correct definition. A good approach to this attribute definition can be found in “The Power of Events” of David Luckham [9], where it lists a set of predefined attributes common to all events, used in the RAPIDE-EPL language. The main attributes can be seen in the following table:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the event</td>
</tr>
<tr>
<td>origin</td>
<td>Object in the monitored system whose execution created the event</td>
</tr>
<tr>
<td>thread</td>
<td>Thread in the monitored system that created the event</td>
</tr>
<tr>
<td>timestamp start</td>
<td>Start time of the event</td>
</tr>
<tr>
<td>timestamp end</td>
<td>End time of the event</td>
</tr>
<tr>
<td>causality</td>
<td>References to immediate predecessors in the causal history of the event</td>
</tr>
<tr>
<td>point-of-creation</td>
<td>Agent in the monitored system that created the event</td>
</tr>
<tr>
<td>trigger set</td>
<td>References to events that were aggregated to create this event</td>
</tr>
</tbody>
</table>

Table 3.5: Event attributes

These attributes must be common for some of the entities defined in the previous section: MeasurementSensor, Configuration, Symptom and Fault. Scalability of some of the entities can derive into specializations of some of them, for example, the symptoms and faults will be from different types, deriving in the declaration of more specific types of symptoms (or faults) (e.g. CPUOverload or VirtualMemoryHigh as Symptom or Orphan-Thread or NullPointerException as Fault).

Other entities (RootCause, Solution, Configuration, Workaround, Patch and PreventiveMeasure), will be originated at server side, but most of them will share these attributes, in order to follow the event structure. One of the main differences will be the origin, which will not be present at the monitored system side, but on the server side instead.

The general and specific purposes defined in section 2.1, are the key for the elements to be defined in the FastFix DSL. Since the implementation depends on the rule engine to be used, and on the combination with the ontologies (for the domain knowledge related sentences), we have only focused on the sentences to be used, without their specific implementation. The result of the evaluation of the elements that event correlation will need is presented in the following tables. The combination of some of the presented sentences will allow the identification of most of the situations related with the software monitoring domain. The nature of this domain, the relationships between its elements, and the general and specific purposes defined in previous sections are the inner spirit of the DSL specification. The DSL elements will tend to grow as the project advances, but here we define the core of the DSL itself, by specifying the main concrete language elements.

First, we are going to list some general sentences associated with fault detection, fault prevention and pattern matching. These sentences are shown in the following tables, first starting with sentences suitable to match for conditions (rule antecedent) in 3.7:
D4.2 Domain Specific Language for Event Correlation

### Antecedent sentences

- [when] from any of the current symptoms
- [when] from any of the current sensor measurements
- [when] from any of the current faults
- [when] -is a known symptom
- [when] -is a known fault
- [when] -is a known sensor measurement
- [when] -is a known performance issue
- [when] -is a known fault-to-happen pattern
- [when] -exists any known fault associated with the current symptoms
- [when] -exists any known solution associated with the fault \{Fault\}
- [when] -exists more than \{count\} symptoms of the fault \{Fault\}
- [when] current level of monitoring of sensor \{Sensor\} is lower than \{Enum\}
- [when] current level of monitoring of sensor \{Sensor\} is higher than \{Enum\}

Table 3.7: General DSL antecedent sentences for the FastFix domain

In 3.9, we present some sentences to be used in the consequent of the rule, in other words, actions to be taken once a fault is detected, if a symptom has been detected as part of several measurements of the sensors, or if there is any known solution or workaround to a previously detected fault:

### Consequent sentences

- [then] report fault \{Fault\}
- [then] report detected symptom \{Symptom\}
- [then] report performance degradation \{PerfDegradation\}
- [then] add attribute \{solution\} with value \{solutionValue\} to the issue with ID \{issueID\}
- [then] add attribute \{workaround\} with value \{workaroundValue\} to the issue with ID \{issueID\}
- [then] add attribute \{description\} with value \{descriptionValue\} to the issue with ID \{issueID\}
- [then] set level of monitoring of sensor \{Sensor\} to \{Enum\}
- [then] get source for current symptom

Table 3.9: General DSL consequent sentences for the FastFix domain

The identification of low level situations (at Event level), correlation of events happening in the same machine, or with other common attributes will be also needed in FastFix. In order to do that, we have followed one of the DSL guidelines about language realization, which is the reuse of existing languages. Some of the following sentences are already in use in some security DSL definitions used in some Drools environments, and they can be helpful to identify those low level situations:
D4.2 Domain Specific Language for Event Correlation

# Antecedent sentences

[when] {Event} repetitions greater than {count}
[when] {Event} repetitions greater than {count} with the same {key}
[when] {Event} repetitions less than {count}
[when] {Event} repetitions less than {count} with the same {key}
[when] Attribute value {key} of {Event} greater than {threshold}
[when] Attribute value {key} of {Event} less than {threshold}
[when] Attribute value {key} of {Event} equal to {threshold}
[when] {EventA}-{EventB} Pair where key is {key}

Table 3.10: Event DSL antecedent sentences for the FastFix domain

# Consequent sentences

[then] {Event1} is an aggregation of {Event2} and {Event3}
[then] modify attribute {key} of {Event} to {value}
[then] insert {Event} in the working memory

Table 3.12: Concrete DSL consequent sentences for the FastFix domain

Causality

As we have already advanced in section 1.2.3, special attention must be paid to causality in the FastFix DSL specification, since we must define structures to allow the detection of possible cause-effect relationships between the events, based on the domain knowledge. Thinking on the rule antecedent defined in section 2.1.2, we need some language structure to check if the type of current events have any causal relationship with recent events, based on the gathered knowledge. Hence, once this condition is true, we must establish the causality connection between these events, using the causality attribute, which references to immediate predecessors in the causal history of the event. This causality attribute can be a list of events that are associated to the creation of the event. Once the knowledge base (or ontology) in the antecedent part of the rule has matched the relationship between an effect and a cause (with the information previously gathered in the knowledge base), we can provide in the consequent part of the rule the relationship of causality in the current events, in other words, the causality chain of the events (with a sequence primitive), connecting the ids of the events of the causality chain with the causality property. Hence, the use of the “sequence” primitive can be really useful within this context, since it helps to establish the mentioned causality connection, which can be implemented as a vector of event identifiers, as we can see in the following figure:
D4.2 Domain Specific Language for Event Correlation

Figure 3.1: Causality chain

# Antecedent sentences
[when]{Event} is a known cause of {Event}
[when] - exists any known cause associated with the fault {Event}
[when] {Cause} is root of causality chain

Table 3.14: DSL antecedent structures for FastFix causality support

# Consequent sentences
[then] {Event1} is followed by {Event2}
[then] add root cause to the issue with ID {issueID}
[then] is the root cause of the fault {Fault}

Table 3.16: DSL consequent structures for FastFix causality support

All of the sentences can be affected with what we will call modifiers, since they can modify the meaning of the actions and conditions with time constraints, logical operations and expressions to identify complete or partial results:
D4.2 Domain Specific Language for Event Correlation

<table>
<thead>
<tr>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>[condition] and [condition]</td>
</tr>
<tr>
<td>[condition] or [condition]</td>
</tr>
<tr>
<td>not [condition]</td>
</tr>
<tr>
<td>[condition] xor [condition]</td>
</tr>
<tr>
<td>at {timestamp} with a range {seconds}</td>
</tr>
<tr>
<td>after {timestamp}</td>
</tr>
<tr>
<td>during {timestamp1} and {timestamp2} over a time window of {count}</td>
</tr>
<tr>
<td>minutes</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>any</td>
</tr>
<tr>
<td>current</td>
</tr>
</tbody>
</table>

Table 3.17: Modifiers

Once we take a look to the sentences needed for the domain, we can notice that some of the structures are replicated, as well as some action elements with a meaning by themselves, which allows the implementation of the scalability mentioned in section 2.3. For example, the following elements are replicated in some of the built sentences: any, current, known, report, add attribute, exists, “to the issue with ID”. Hence, these elements can be defined independently, as well as the operators and other modifiers.

Regarding the “known” element, it is strongly related with the connection to the knowledge base, a concept that we are currently investigating through a possible integration of the rule-based system and description logic reasoners. Code implementation of some of the sentences with the word “known”, can be implemented as high-level calls to the reasoner, to perform the domain knowledge “queries”. Hence, sentences like: “any known fault associated to the current symptom list”, which would be expressing the connection at reasoner level. As an example, a possible implementation in Java code of this sentence can be as follows:

```java
[when] -any known fault associated with the current symptoms = $f : Fault() from OwlGlobal.getFaultList($simptomList)
```

Finally, because of the FastFix rationale and architecture, the event correlation component must connect with other components to gather monitored information from the sensors (Context Observer), or calling for specific actions to other components, like requesting an increase or decrement of the level of monitoring (Context Observer), depending on the probability of a possible fault being raised (or because of the detection of a current performance degradation). Moreover, other actions can be requested, like the proposal of a control objective to the patch generation component, once a the pattern of a new fault has been recognized, or the request for recording to the fault replication component, once a fault that has not occurred yet is considered as possible.
3.4 Conclusion

FastFix rules will be focused on identifying situations, detecting faults from symptoms, as well as preventing an imminent fault, or identifying all the information associated with the fault, like the cause that led to the fault to happen, as well as their solutions, or possible workarounds. Thus, the FastFix DSL creation must allow to express all the conditions and actions related with these situations. With that purpose, the sentences included in the table 3.7 provide a way to define these conditions and possible consequences. Fault prevention is made possible with the identification of the occurrence of any, all or more than a specific number of symptoms of an specific kind of fault.

Some of these sentences are associated with the concept of knowledge, since they require a mechanism to describe what is the behavior of the domain, as well as some relationships between software faults or performance degradations and their possible causes. As we mentioned in section 3.1.1, we are investigating possible code implementations of these
knowledge related sentences, trying to define a conceptual model, with the usability and reuse always in mind.

In order to properly complement these expressions, some modifiers have been defined, allowing the representation of time concepts, number of repetitions, identification of events with a common value in one of its properties. Other general situations can be expressed, like the occurrence of any type of event while another event happens, with the use of temporal modifiers and the use of the pair primitive and/or pairing them with the common value of an specific property.

One of the features FastFix will implement is the error reporting, which will show a list of the current error (or tickets) reported by the system, as well as some information regarding the fault, like an issue tracking system. In order to express the actions to be done regarding error reporting, some structures have been proposed, attaching different kind of information related with the error.

Causality will play a prominent role in most of event correlation scenarios, hence the possibility to create and grow causality chains is a must. The use of the causality property in any of the events and the use of a knowledge mechanism like ontologies makes things easier.

Finally, the request for communication with other FastFix components, like the request for control objectives for the patch generation component, the request for increasing or decreasing the level of monitoring to the context observer and the alarm triggering for recording to the fault replication component are good examples of the expressiveness needed to connect them, even from the rule side.
Bibliography


