D2.3: User Requirements and Conceptual Architecture

Lead contractor: TUM
Dennis Pagano, Tobias Roehm, Benoit Gaudin, Emitzá

Author(s): Guzmán, Sergio Zamarripa, Walid Maalej, João García,
Daniel de los Reyes, Alessandra Bagnato, Javier Cano

Submission date: May 2011
Dissemination level: Restricted
**Abstract:** In this document we give an overview about the FastFix approach. We describe concrete scenarios where the maintenance of a software application can be supported or automated by monitoring its execution and context. The document develops a concrete technical blueprint of the envisaged FastFix system.

In the first part, we present the requirements for the FastFix system that were derived by interviewing and observing maintenance engineers and analyzing the desired functionality of the FastFix system. This analysis covers three aspects. First, we give a classification of error types, the FastFix system might have to deal with. Second, we describe the FastFix error handling strategies, i.e. the different approaches that handle the specific error types. Third, we introduce concrete scenarios from real world applications of the consortium companies, combining error types, types of target applications, and underlying technologies. With the requirements, we define the scenarios and situations on which we will focus during implementation of the FastFix system.

In the second part we present the high-level architecture of the FastFix system. It is divided into a client and a server part and gives an overview about important components and their interfaces. With the architecture, we describe the structure of the the FastFix system that implements the requirements.

In the third part we introduce the main ontologies of the FastFix system. These ontologies are used as data representation mechanism in the FastFix system. They formalize the conceptual model, i.e. important concepts and their relationships within domains that are relevant to FastFix. With the ontologies, we sketch which data have to be stored and processed to support the requirements and how this will be done.

Finally, we discuss how techniques from the field of artificial intelligence can be applied and combined within the FastFix system in order to meet the requirements.

---

This document has been produced in the context of the FastFix Project. The FastFix project is part of the European Community’s Seventh Framework Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.
Contents

1 Introduction 7

2 Requirements 9
  2.1 Interviews ............................................................................................................. 9
  2.2 Scope of FastFix ................................................................................................... 10
    2.2.1 Error Types .................................................................................................... 12
    2.2.2 Error Handling Strategies ............................................................................ 13
    2.2.3 Scenarios ....................................................................................................... 17
  2.3 Functional Requirements ................................................................................... 26
    2.3.1 User Requirements ....................................................................................... 26
    2.3.2 System Requirements ................................................................................... 29
  2.4 Non Functional Requirements ............................................................................ 31
    2.4.1 General Applicability ..................................................................................... 32
    2.4.2 Performance .................................................................................................... 32
    2.4.3 Extensibility of Context System ...................................................................... 32
    2.4.4 Privacy ........................................................................................................... 32
  2.5 Mockups .............................................................................................................. 32
    2.5.1 Error Report Generation .................................................................................. 32
    2.5.2 Context Augmented Fault Replication ............................................................. 34
    2.5.3 Error Cause Recommendation ....................................................................... 38
    2.5.4 Self-Healing and User Feedback ................................................................... 39

3 Architecture 41
  3.1 FastFix System Overview .................................................................................... 41
  3.2 FastFix Client Platform API ................................................................................ 43
    3.2.1 Client Data Store ............................................................................................. 43
    3.2.2 Context System ............................................................................................... 44
    3.2.3 Client Self-Healing System ............................................................................. 44
    3.2.4 Error Reporting and User Feedback System .................................................. 45
    3.2.5 Application Bridge .......................................................................................... 46
    3.2.6 Communication System .................................................................................. 48
  3.3 FastFix Server Platform API ................................................................................ 48
    3.3.1 Server Data Store ............................................................................................. 48
    3.3.2 Event Correlation System .............................................................................. 48
    3.3.3 Fault Replication System ............................................................................... 49
    3.3.4 Server Self-Healing System .......................................................................... 50
    3.3.5 Maintenance Engineering UI System .............................................................. 51
    3.3.6 Maintenance Environment Bridge ................................................................. 51

4 Ontologies and Conceptual Model 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Reasons For Using Ontologies</td>
<td>54</td>
</tr>
<tr>
<td>4.2 Description of FastFix Ontologies</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1 Overview</td>
<td>56</td>
</tr>
<tr>
<td>4.2.2 Application Execution Ontology</td>
<td>57</td>
</tr>
<tr>
<td>4.2.3 Domain Ontology</td>
<td>61</td>
</tr>
<tr>
<td>4.2.4 User Interaction Ontology</td>
<td>62</td>
</tr>
<tr>
<td>4.2.5 Maintenance Ontology</td>
<td>63</td>
</tr>
<tr>
<td>4.3 Examples of Ontology Usage</td>
<td>67</td>
</tr>
<tr>
<td>4.3.1 General</td>
<td>67</td>
</tr>
<tr>
<td>4.3.2 Input Validation Scenario</td>
<td>67</td>
</tr>
<tr>
<td>4.3.3 Crashing Java Application Scenario</td>
<td>68</td>
</tr>
<tr>
<td>4.3.4 Configuration Scenario</td>
<td>68</td>
</tr>
<tr>
<td>4.3.5 Mobile Scenario</td>
<td>68</td>
</tr>
<tr>
<td>4.4 Usage of Ontologies in FastFix Components</td>
<td>68</td>
</tr>
<tr>
<td>4.4.1 Context System</td>
<td>68</td>
</tr>
<tr>
<td>4.4.2 Event Correlation System</td>
<td>69</td>
</tr>
<tr>
<td>4.4.3 Fault Replication System</td>
<td>70</td>
</tr>
<tr>
<td>4.4.4 Client Self-Healing System</td>
<td>70</td>
</tr>
<tr>
<td>4.4.5 Server Self-Healing System</td>
<td>70</td>
</tr>
<tr>
<td>5 Intelligence in FastFix</td>
<td>72</td>
</tr>
<tr>
<td>5.1 Alignment of Intelligence Technologies to FastFix Requirements</td>
<td>72</td>
</tr>
<tr>
<td>5.2 Possible Combinations of Intelligence Technologies</td>
<td>75</td>
</tr>
<tr>
<td>6 Summary</td>
<td>77</td>
</tr>
<tr>
<td>Bibliography</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Overview of Error Handling Strategies in FastFix............................................. 14
2.2 TXT Production Planning: Database error message ................................................. 23
2.3 Error Reporting User Interface (Mockup)............................................................... 33
2.4 Automatically Generated Error Report on Maintenance Site (Mockup).................. 34
2.5 Context Augmented Fault Replication (Mockup)..................................................... 35
2.6 Context Augmented Fault Replication - System Context (Mockup) ....................... 36
2.7 Context Augmented Debugging (Mockup).............................................................. 37
2.8 Error Cause and Error Report Recommendation (Mockup)..................................... 38
2.9 Self-Healing Using Control Objectives (Mockup)................................................... 39
2.10 User Feedback Interface (Mockup)......................................................................... 40

3.1 FastFix Client Platform Overview............................................................................ 42
3.2 FastFix Server Platform Overview.......................................................................... 43
3.3 Interfaces of the Client Data Store......................................................................... 44
3.4 Context System Interfaces....................................................................................... 44
3.5 Client Self-Healing System Interfaces..................................................................... 45
3.6 Error Reporting and User Feedback System Interfaces.......................................... 46
3.7 Application Bridge Interfaces.................................................................................. 47
3.8 Event Correlation System Interfaces....................................................................... 49
3.9 Fault Replication System Interfaces....................................................................... 50
3.10 Server Self-Healing System Interfaces.................................................................... 51
3.11 Maintenance Engineering UI System Interfaces..................................................... 51
3.12 Maintenance Environment Bridge.......................................................................... 52

4.1 Overview of domains covered by FastFix ontologies.............................................. 57
4.2 Definition of application execution context.............................................................. 58
4.3 EnvironmentEvent Taxonomy.................................................................................. 58
4.4 ApplicationEvent Taxonomy.................................................................................... 59
4.5 Application Structure.............................................................................................. 60
4.6 Application Execution Model................................................................................. 61
4.7 Simple SCM Domain Ontology.............................................................................. 62
4.8 User Event Taxonomy.............................................................................................. 62
4.9 User Interaction Domain......................................................................................... 63
4.10 Overview of Conceptual Error Model..................................................................... 64
4.11 Cause Taxonomy..................................................................................................... 65
4.12 Fixing related concepts........................................................................................... 66
4.13 Error Report Model................................................................................................. 66
4.14 Maintenance Process............................................................................................... 67
List of Tables

2.1 Expected Relevance of Error Handling Strategies for Specific Error Types. 17
1 Introduction

FastFix Overview  FastFix is a research project funded by the European Community’s Seventh Framework Program. FastFix will enable time- and cost-efficient maintenance and support services, by monitoring software applications, replicating semantic execution failures, and automatically generating patches. Software maintenance and support services are key factors to customers’ perception of software quality. Software vendors need a system to remotely provide a high quality support service to their customers, improve user experience and facilitate corrective, adaptive and preventive maintenance – of both new and existing software products.

FastFix results will include a platform and a set of open source tools for online monitoring of execution environments, gathering semantic information on application and user behavior. This information is sent in real time to a support centre, taking special care on privacy and security issues. Using event correlation techniques, FastFix identifies failure symptoms, performance degradation or changes in user behavior and allows for failure replication, patch generation and patch deployment, leading to self-healing software applications.

Four main lines of research are fundamental for the FastFix project and constitute the core of innovation activities. Context elicitation and user modeling determine which and how information on execution and interaction is going to be gathered independently from the application and its environment. Event correlation determines how the gathered information is going to be used to draw conclusions about the kind of problems the application is facing and what possible causes are. Fault replication provides the platform that allows replicating errors as close to the real circumstances as possible. Patch generation and self-healing determines which and how patches are going to be generated, and how they will be applied to the application at the runtime environment.

Main objectives are to develop (1) tools to gather context information on user and application, (2) a run-time with minimum impact on application performance, (3) a secure method to send this information to a centralized fault analysis platform, (4) a tool to detect software failures, undesirable execution trends and performance degradation, (5) a platform to replicate failure conditions within a virtual machine and (6) a tool to generate change strategies and necessary patches.

Document Overview  The purpose of this document is to give an overview of the FastFix approach and develop a concrete technical blueprint of the envisaged FastFix system. The focus is on issues that are on a system-wide scale and we especially address issues that are relevant to more than one FastFix work package or component.

More specifically, this document defines requirements for the FastFix system. The requirements define which functionality will be provided by the FastFix system and which paradigms and approaches are used. We present an overview of the overall FastFix system and describe the cooperation required between components to implement global function-
ality. As an important contribution of this document we present the FastFix error handling strategies as part of the requirements. We further include concrete scenarios and mockups to give a first impression of supported functionality and how real results may be presented to the maintenance engineer.

Further, we sketch the architecture of the FastFix system and discuss important components and their interfaces. The architecture provides the basis for the implementation of FastFix components and their integration to a functional system.

We discuss further the management of shared data, an important integration aspect. We define ontologies that store relevant data that is shared between different components within FastFix. They can be used by a specific component in two ways, either by storing information in the format provided by the ontologies or reading and processing data stored in form of ontologies.

Finally, the description of different techniques from the field of artificial intelligence serves as a reference frame to think about how the requirements that demand intelligence capabilities like reasoning and recommendation can be implemented.

This document serves as architectural and conceptual reference framework during the implementation and development of the FastFix system. Concepts and components described in this document will be detailed and implemented in the different work packages and described in the corresponding deliverables of each work package.

**Document Structure**

This document is divided into four parts.

After this short introduction, we present the requirements for the FastFix system that were derived by interviewing and observing maintenance engineers and analyzing the desired functionality of the FastFix system. In Section 2.1 we summarize results of interviews conducted with maintenance engineers and in Section 2.2 we discuss factors influencing the scope and applicability of the FastFix system. The analysis of desired FastFix functionality covers three aspects. First, we give a classification of error types, the FastFix system might have to deal with. Second, we describe the FastFix error handling strategies, i.e. the different approaches that handle the specific error types. Third, we introduce concrete scenarios from real world applications of the consortium companies, combining error types, types of target applications, and underlying technologies. Further, we summarize the functional requirements and non functional requirements of the FastFix system. The mockups shown in Section 2.5 give a first idea how user interfaces of the FastFix system might look like.

In the second part we present the high-level architecture of the FastFix system. The architecture is divided into a client and a server part and we give an overview about important components and their interfaces.

In the third part we introduce the main ontologies of the FastFix system: the Application Execution Ontology, Domain Ontology, User Interaction Ontology, and Maintenance Ontology. These ontologies are used as data representation mechanism in the FastFix system. They formalize the conceptual model, i.e. important concepts and their relationships within domains that are relevant to FastFix.

Finally, we discuss in Section 5 how techniques from the fields of Artificial and Computational Intelligence, namely rule-based systems, ontologies and machine learning, can be used and combined within FastFix to fulfill the requirements.
2 Requirements

This chapter summarizes requirements for the FastFix system. The requirements were collected through interviews with maintenance engineers from the industrial consortium members (2.1) discussion among the consortium and analysis of the desired functionality of the FastFix system. They serve as orientation of the functionality the FastFix system should provide.

2.1 Interviews

In order to elicit requirements for FastFix we conducted interviews with maintenance engineers, the main FastFix stakeholders. We summarize the results of the interviews in this section.

Interview setting

The goal of the interviews was to understand the work routine of maintenance engineers, identify their problems and needs and collect first ideas how these can be addressed by the FastFix system. We conducted face-to-face interviews with 4 software maintainers from consortium companies in October 2010. Each interview lasted for 3 hours. They were split into three parts dealing with state-of-the-art practices in software maintenance, possible solutions and improvements, and requirements for the FastFix system. The interview was focused to these parts by using a questionnaire. The interviewers were not forced to follow the questionnaire question by question, but they used the questionnaire as structuring element. The statements and problems mentioned during the interviews where quite overlapping. This gives a good indication that the results can be generalized.

State of the Practices in Software Maintenance

A typical workflow of maintenance engineers when solving a ticket is as follows: The first task is to gather necessary information like build version of the software. Then, they try to reproduce the error in a local system. After that, they download the latest version of source code from a version control system and modify the source code in order to handle the ticket. To test the solution, maintenance engineers then build a patch, install it on a local system and perform some basic tests. If these tests succeed, they commit the modified source code to the version control system and close the ticket.

Frustrating and long lasting tasks most maintenance engineers agreed on were “finding solutions to similar problems”, “reproducing errors” and “understanding the cause of an error”. No interviewee reported a satisfying approach for the issue of “finding solutions to similar problems”. Workarounds and best practices to deal with the issue are to search in ticketing systems and commit messages for similar problems using keyword search. “Reproducing errors” was rated difficult because it is time consuming, demands to reconstruct
the same environment as the users’ and necessary information is often not available. Similar to that, “understanding the cause of the error” was judged difficult because getting enough information is difficult, especially in case of third-party components.

The interviewees said they need the following information to be able to reproduce and understand an error: current system configuration (operating system, software release version), current hardware configuration, log files, stack trace, user actions that led to the error and current state of the system (e.g. CPU load).

Information often missing was identified to be documentation of third-party libraries or APIs, source code for debugging, information about user actions that lead to an error, specific problem definitions and knowledge about symptoms and causes of an error and the relations between them.

Problems in software maintenance are according to the interviewees reproducing an error (problem of reproducing the same environment), getting information about the actions of the user that lead to an error and handling third party software (because code cannot be debugged and it is often unclear how the third party software should be used correctly).

Possible Improvements in Software Maintenance

We asked the interviewees to envision which tools or changes can improve their daily work and got the following results:

- One idea that was reported several times was to have a central information dashboard collecting information about “user procedure, possible failures and solutions and the environment”. It should not exhibit a pull-fashioned approach like forums and should allow to access information at different granularity levels and hide irrelevant information.

- Providing a knowledge base covering several aspects of the system under inspection is another idea that was reported several times. More specifically, the knowledge base should cover symptoms, causes, errors with classification, possible solutions and people with expertise and provide “where to look for”-pointers as starting points when looking for the cause of an error or searching for a solution.

- Another issue that was reported was the fact that information from different sources like commit messages and tickets is not linked but linking them together would bring a benefit of easier discovery of related and relevant information.

- Improving communication with users is a concern of the interviewees to get more information from them, especially about the actions a user performed before an error occurred.

- Finally, support for managing dependencies between components would be helpful, e.g. when doing an impact analysis of the effect of a change.

2.2 Scope of FastFix

Software maintenance techniques in general depend on many factors that influence and limit their applicability and relevance. FastFix is a complex framework that aims at
supporting remote software maintenance by providing a platform and a set of strategies that handle errors under specific circumstances. Though the platform is designed in a general and extensible way, the following factors influence the functionality scope of the developed system.

- **Error type.** In practice, the types of errors that may occur in a system are not limited. FastFix includes strategies to handle several important error types (see Section 2.2.1) in the first integrated version, and is designed as an open system that allows for future extension of these strategies and their applicability for specific errors.

- **Error handling strategy.** Errors can be handled in different ways, and these might change based on future research. In the FastFix project, we identified ten strategies that reflect maintenance engineering needs as well as state of the art in research. Error handling strategies may be added in future versions.

- **Application type.** In particular the architecture of the target application imposes constraints on which strategies may be applied to solve given errors. Standalone desktop, client-server, web, and mobile applications all provide different levels of complexity.

- **Implementation technology.** The implementation technology used to build the application influences how it can be maintained. Examples for different implementation technologies include native Java, Eclipse RCP, and Ajax applications.

- **Source availability.** Modifications to an application can be done in different ways. The source code availability of an application influences the degree of freedom when it comes to change faulty behavior. Further, many monitoring techniques require sensors to be built into applications. Different availabilities include open source applications, bytecode availability, and closed source applications. Moreover, many applications rely on closed source third-party components.

- **Relevance.** Business and research potential influence error handling in a different way. The FastFix system should be competitive in the business world and solve problems that are relevant in practice. Moreover, the occurring problems should be of relevance in research.

The described scope factors influence the functionality of FastFix, as well as the effort required to implement this functionality. For example, when fixing a closed-source application it is not possible to change the source code and recompile the application to solve a given problem. It is thus necessary to target functionality under specific circumstances in the first integrated version of FastFix.

The FastFix system will facilitate remote software maintenance. To this end, it has to provide generic means that may handle specific errors in different ways. In the following, we describe the different errors in the scope of FastFix (Section 2.2.1). Then we introduce the FastFix strategies to handle the important error types (Section 2.2.2). Finally, we describe four main scenarios (Section 2.2.3) that define the specific application types, implementation technologies, source code availability, which we will target in the first integrated version of the FastFix system.
2.2.1 Error Types

In this section we give a short description of different types of errors that may occur during execution of an application and consequently the FastFix system may have to deal with them.

**Application Crashes** The term “application crash” refers to a common symptom and means that an application is closed by the runtime environment due to an error that was not handled by the application itself. The causes for this symptom can be manifold. Typical causes include unhandled exceptions or threading issues like deadlocks and race conditions. Application crash errors may manifest in thrown exceptions, freezing application or closing application. We address application crashes in the scenario described in Section 2.2.3.2.

**Configuration Errors** A “configuration error” occurs when an instance of an application has wrong configuration parameters and this leads to a deviation of intended behavior. Further, the environment of the application may be configured wrongly or the configuration of the application and the environment may be inconsistent. Symptoms of configuration errors can be different, ranging from having no effect at all to causing the application to crash. Configuration errors typically depend on input data such as configuration files (which may or may not be unknown to the user) or parameters passed to the application at startup. The scenario described in Section 2.2.3.3 addresses configuration errors.

**Dependency Errors** “Dependency errors” result from 3rd party components like libraries or web services that are used by an application. When these components are not installed or a wrong version of them is installed, errors can occur that are caused by wrong invocation (wrong syntax or usage in a wrong way) and that may lead to wrong functionality or exceptions. Of course 3rd party components may have their own errors (even when invoked properly) and if such an error occurs in the 3rd party component it is propagated to the application using the component. The scenario described in Section 2.2.3.4 addresses dependency errors.

**Hardware Errors or Resource Errors** These errors are related to hardware or resources. Causes for this type of errors can be manifold like the application is consuming too many resources or competing with other applications for the same resources. Symptoms of the errors are slow performance, long waiting time or an application crash.

**Input Validation Errors** “Input Validation Errors” are errors that occur if a user enters – purposefully or casually – input that is not expected by the application and consequently not handled. Symptoms can range from having no effect via exhibiting wrong functionality to data loss (vulnerability). The scenario described in Section 2.2.3.1 addresses input validation errors.

**Application Interference Errors** “Application Interference Errors” occur when two or more applications are installed or run on a system and interfere with each other. The interference does not include mutual invocation or cooperation (errors occurring in these situations are treated as dependency errors) and hardware-related issues (errors occurring
in these situations are treated as hardware errors or resource errors). One possible cause is if both applications partly share the same configuration and one application changes it and the change causes the other application to crash or exhibit wrong behavior.

**Usability Errors** We denote the situation of a user not being able to use the software in a smooth and comfortable way as “usability error”. Usability errors are difficult to handle, because it is hard to clearly define usability and to recognize usability errors. Causes for usability errors include flaws in user interface design such as information overload, lack of information, or unclear navigability. Symptoms of usability errors include specific user interaction patterns. Users might for instance stop to use the application, or click randomly. Others might be inactive for a relatively long time period or search for a tutorial describing the application usage. The scenario described in Section 2.2.3 addresses usability errors.

### 2.2.2 Error Handling Strategies

The FastFix system will support the handling of errors in maintenance in the following ways, depending on concrete scenarios as described in Section 2.2.3. First, the FastFix system will enable the reproduction of the error on the remote maintenance site. To this end, it has to continuously record the application execution during the runtime of the target application. The recorded data has to be obfuscated and sent to the FastFix server to be further processed. Second, the FastFix system will provide additional context information to software maintenance engineers during error reproduction. Last, if possible, the FastFix system will prevent the error from happening by self-healing the application using functionality degradation.

FastFix employs different strategies to handle a specific error type depending on its complexity and scope. First, error complexity measures how difficult an error is to spot, to understand, and to resolve. Thus the complexity of an error depends on the specific symptoms and causes of the error. This does not necessarily impose that different error types have a different degree of complexity. The second dimension describes properties of the environment where an error occurs (cf. Section 2.2). This includes application architecture (e.g. client-server), implementation technology (e.g. Eclipse RCP), and the availability of the application’s source code (e.g. open source). Both error complexity and scope influence the applicability of an error handling strategy, and how difficult it is to accomplish.
Error Report Recommendation  The simplest strategy to handle an error is to report it to the maintenance site. To this end, FastFix provides a common framework for reporting errors even for applications that do not include such a mechanism. The FastFix system may recommend an error report to the end user when an error occurred. It will consider whether an error report is adequate and unobtrusive to the user at a current point in time. The FastFix system will additionally provide means to send the error report to the maintenance site including as much helpful information as possible to facilitate error reproduction.

Error Report Generation  Error reports usually lack important information needed to understand and reproduce errors on the maintenance site \[10\]. Error report generation in the FastFix system aims at automatically creating error reports that contain this relevant information. The FastFix system automatically collects the steps to reproduce the error in the report. Additionally, error reports will include context information collected by
a component that observes user interaction, application execution, and other relevant aspects of the target environment.

**Fault Replication**  In order to analyze application faults, its execution has to be deterministically replayed on the maintenance site. Replaying application execution aims to understand what led to faults and provides the basis for the work of maintenance engineers. To enable automatic execution replaying, the FastFix system utilizes context information gathered by the context elicitation components. Further, the replay functionality provides facilities to be used interactively by maintenance engineers. To this end, engineers may move forwards and backwards within a video-like reproduction of the application execution.

As a prerequisite for the replication of application execution, data about the execution will be collected and transferred to the maintenance site. In order to guarantee privacy, the collected data will be obfuscated.

**Context Augmented Fault Replication**  Today error reports widely differ in their quality [10]. In the best case they contain static information about the application execution on the end-user site and manually entered steps to reproduce the error. Maintenance engineers then interpret this information and try to reproduce the fault to be able to identify its causes. The occurring gap between the application execution and the execution replay on the maintenance site represents a strong threat to success when reproducing errors.

As described in the paragraph on error report generation above, the FastFix system will generate error reports and augment them with additional context information about application execution, user interaction, and runtime environment. These error reports form the base for context augmented fault replication. The FastFix system provides maintenance engineers with additional context information while they are replaying an erroneous situation.

The according information will be represented in a graphical way to provide easier overview for maintenance engineers. The FastFix system will further use statistical analysis to aggregate information and provide useful statements in different levels of granularity.

**Error Cause Recommendation**  Upon the receipt of error reports and by reproducing faults accordingly, maintenance engineers try to identify the causes of errors. The FastFix system will provide maintenance engineers with recommendations on possible causes of specific reported errors. To this end, the FastFix system will use statistical analysis and event correlation techniques to derive error causes from similar reported symptoms.

**Error Cause Explanation**  When the cause of an error is known, it can be helpful to inform end users about it. The FastFix system provides a user feedback mechanism that allows to inform the end user of target applications about possible causes. Thus, even applications without user interfaces may deliver feedback to their users.

**Solution Recommendation**  The FastFix system supports maintenance engineers by recommending solutions to issues based on their causes. The system uses correlation techniques to infer possible solutions and relevant source code parts based on similar causes and symptoms for which solutions already have been developed.
Fix Recommendation  The FastFix system will also be able to recommend corrective actions to end users. As described in the paragraph on error cause explanation above, this will be possible due to a feedback channel within the framework. Fix recommendations may include workarounds or solutions that may not be automatically applied by the FastFix system. Further, the FastFix system may also provide facilities for the user to model the undesired symptoms. This model can then be used in order to automatically generate a patch.

Patch Generation  In order to tackle and remove undesired symptoms, the target application has to be changed in some way. Such a change is called patch and examples for patches are a class file with modified byte code, a file containing a new supervisor model or a configuration file storing the correct configuration of the application. The feature of patch generation within the FastFix system denotes the ability to create patches in order to modify the application behaviors. An alternative to this approach is to retrieve the patch from a patch store.

Self-Healing  Self-healing denotes a property of a system to automatically recover from errors occurring within the system. To achieve that, a patch has to be generated that removes issues from the system and has to be applied in an automated way. Consequently, patch generation is part of self-healing.

Within the FastFix system, the self-healing component prevents the target application from reaching improper states. It contains a supervisor which observes the target application executions and prevents from undesired executions. This supervisor makes decisions based on a model of the application behaviors. When some properties need to be ensured on the target application, a new set of proper behaviors, called a patch, can be provided. In some situation, the self-healing component is also able to detect the presence of an error and trigger a mechanism for the automatic generation of a patch.

The FastFix self-healing approach automatically adapts to changes in the application. If a new version of an application is released or if a new application needs to be maintained, then the self-healing component may automatically adapt and add autonomic features to it (automatic model extraction, automatic sensor deployment and automatic actuator deployment, i.e. control points in the program).

Comparison  Whether an error handling strategy can be successfully applied to a specific error depends on particular factors like the application type, implementation technology, and source code availability. In this section we compare the FastFix error handling strategies in terms of their expected relevance given specific error types. Table 2.1 shows the error handling strategies (rows) that we expect to be suitable to handle errors of specific types (columns).
Table 2.1: Expected Relevance of Error Handling Strategies for Specific Error Types.

<table>
<thead>
<tr>
<th>Error Handling Strategy</th>
<th>Application Crash</th>
<th>Configuration</th>
<th>Dependency</th>
<th>Hardware or Resource</th>
<th>Input Validation</th>
<th>Application Interference</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Report Recommendation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Error Report Generation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Fault Replication</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context Replication</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Augmented Fault Replication</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Cause Recommendation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Error Cause Explanation</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution Recommendation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Fix Recommendation</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patch Generation</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Self-Healing</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the case of input validation errors, an error handling strategy that generates and applies a patch to the faulty application is expected to be most relevant. Section 2.2.3.1 describes how a patch may be generated in order to prevent malicious user input from being processed.

In order to handle application crashes, we expect the generation of an error report and the replication of the fault on the maintenance site to be most relevant. Section 2.2.3.2 describes a concrete scenario that illustrates this.

Configuration errors can often be resolved by the user. Beneath the generation of an error report, we therefore expect a fix recommendation to the end user to be a relevant error handling strategy. Section 2.2.3.3 illustrates this case with a concrete scenario.

Finally, usability errors may be observed and an according error report may be automatically generated and sent to the maintenance site without user interaction. The created awareness of usability issues as well as recommendations of guidelines should help developers to fix the errors. We describe the solution recommendation strategy in the scenario in Section 2.2.3.4.

### 2.2.3 Scenarios

In the following we describe four scenarios which show the functionality and mode of operation of the FastFix system in detail. The scenarios resulted from an extensive scope discussion in the FastFix consortium as well as a detailed analysis of the industrial partners’ needs and the related work. We use these scenarios during the implementation and evaluation phase as a common baseline of functionality. However, we might also refine
and extend them if necessary. Each scenario describes the target application concerned (i.e. where does the error occur), the cause (i.e. why does the error occur), the symptoms (i.e. what do users or developers perceive), as well as the error handling strategy applied by the FastFix system.

2.2.3.1 Input Validation Error – Main Strategy: Self-Healing

**Error description** SQL injection is a technique that exploits a security vulnerability in the database layer of an application by injecting TRANSACT SQL code into textual user input or HTTP POST data streams. This vulnerability affects web applications based on dynamic languages like PHP or ASP. The security error occurs when user input is incorrectly validated and thus input containing SQL statements is forwarded to and processed by the database system. In general, such vulnerabilities occur whenever one programming or scripting language is embedded inside another.

In this scenario we examine Webgoat 5.4, a “deliberately insecure J2EE web application designed to teach web application security lessons” maintained by the OWASP project. Webgoat is provided as open source (GNU GPL v2). As Webgoat was developed to provide a testing environment for security vulnerabilities it is well suited as prototype application to show the functionality of the FastFix system.

**Symptoms** In case of a SQL injection, confidential data is presented to an unauthorized intruder or data is deleted or altered in the database. The first case exhibits erroneous behavior because the information is confidential and can be misused if publicly accessible (e.g. selling contact information, credential stealing etc.). In the second case, the developers either remain unaware of the error or discover it by receiving other error reports of software components that try to access data that was deleted or altered because of the SQL injection attack. In our scenario, we focus on an intruder trying to access user information like credit card numbers.

**Cause** The cause of an SQL injection is an attack by a malicious user. The attack is usually executed in two steps. First, intruders tests if they can get access to database system output like error messages by entering an SQL string that causes a database error into the data stream of a HTTP POST. An example for such malicious POST data is “employee_id=101’and 1=1--&password=121&action=Login”. This string is forwarded by the web application to the database system, which executes the malicious TRANSACT SQL and displays the resulting error message to the user. In the second step, intruders perform the actual attack by now posting an SQL string that either displays, deletes or alters data in the database. An example for such a malicious POST data stream is “employee_id=101&password=121'+union+select+from+passwords+*--&action=Login”. This string is also forwarded to the database system, processed there and the result is displayed to intruders who have achieved their goal to access user information.

**AS-IS** The following items describe the state of the art procedure to handle SQL injection attacks.

---

1 https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
2 https://www.owasp.org/
• **Error Reproduction**
  When developers discover the data loss they manually investigate the log files. Experienced developers will then recognize the malicious SQL statement. To identify which feature in the application is not validating the input, developers have to check the timestamps of the SQL statements, the corresponding user sessions, and the HTTP request log. If the affected feature is identified, usually by detailed search, the developer might create a bug report.

• **Error Handling**
  Developers locate where in the code the input validation should be performed. They add code that parses and validates the user input and rejects input containing SQL injection, test the changes, commit them, and close the bug report. The developers create an update of the affected software by hand and release it to prevent other instances of the application from similar errors.

**TO-BE** In the following, we describe how the FastFix system handles input validation errors, including the initial detection and the events created as well as the deployment of the final patch.

1. **Observe first malicious request**
   As the FastFix system is capturing *user inputs*, it captures the string which is actually a SQL injection. Using textual pattern matching, the FastFix system can detect that an SQL injection is taking place and creates the event **e1** “SQL Injection Attempt” which is a subtype of “Invalid Input”.

2. **Observe application reaction**
   The FastFix system also captures the reaction of the target application which is the presentation of an error message to the user. This results in the creation of event **e2** “Backend Error Reported To User”. This step is necessary to discriminate between system in which SQL injection is already handled and rejected. It requires the FastFix system to be able to detect if a database error message is shown to the user.

3. **Increase sensing level automatically**
   As both events are submitted to the FastFix server, the FastFix system recognizes that it is very likely that an SQL injection attack is happening and it increases the level of detail in logging and monitoring in order to acquire more data by signaling the corresponding components (i.e. Tomcat Server, JRE, DB server and FastFix sensors).

4. **Observe second malicious request**
   Like the first SQL injection string, the second malicious request is also detected by the FastFix context system. As the execution of this SQL string causes an error in the database, this error is detected by inspection of the database log (which is now much more detailed) and a third event **e3** “SQL ERROR DETECTED” is created.

5. **Detect attack**
   As all the generated complex events match the following rule “**e1e2e3** -> Successful SQL Injection” the event correlator detects a successful SQL injection.
6. **Retrieve or create patch**
   In order to remove the cause of the problem, the Event Correlator requests a *patch* from the Patch component. If a patch already exists, it can be retrieved and applied (maybe small modifications to adapt the patch to the current context may be required). In this case, the patch generator creates an execution model which prohibits the target application to reach the “SQL injection” state. If no patch can be generated or retrieved, the FastFix system recommends a workaround by activating an Intrusion Prevention System in order to block this kind of traffic.

7. **Apply patch**
   At the end, the Patch Component provides a patch to the FastFix client, which applies it by controlling the application through a supervisor component. The user and the developers are notified accordingly.

2.2.3.2 **Java Application Crashes – Main Strategy: Fault Replication**

**Error Description** A Java application crash occurs when a Java application that runs in the Java virtual machine is terminated unexpectedly because of an error. Causes for such crashes can be manifold with typical causes including unhandled exceptions, runtime environment errors, threading issues, and memory shortage. In this scenario we examine application crashes in MOSKitt\(^3\), a free open-source software modeling tool developed by the FastFix consortium member Prodevelop. MOSKitt is developed within the Eclipse ecosystem reusing some other open source Eclipse plugins. As MOSKitt is an open source project, the source code is available to the FastFix system. We will use an earlier version of MOSKitt that has known application crash problems as case study.

We focus on a situation in which a user is working with MOSKitt editing a graphical diagram in one of the MOSKitt editors, which are based on EMF and GMF.

**Symptoms** When the user is right clicking on the editor to bring up the context menu and selecting one of the submenu entries, the 'Navigate' entry concretely, the jvm crashes and closes with no previous warning or error message.

**Cause** The cause of the application crash is a segmentation fault. The error occurs only on Linux machines. If the application is launched from a console, which usually not done by a typical user, the error message shown in Listing 2.1 is printed giving a hint about the problem.

**AS-IS** The following items describe the state of the art procedure to handle application crashes.

- **Error Reproduction**
  On the maintenance site, maintenance engineers try to reproduce the crash based on the error reports from users and stack traces. This is a hard problem as it is difficult to restore the same environment as the one on the user’s machine and often the information provided in the error report is not sufficient enough to exactly reproduce the user’s actions that were leading to the crash. In our scenario, typical

\(^3\)http://www.moskitt.org/eng/
Listing 2.1: MOSKitt: Error message printed to console

# A fatal error has been detected by the Java Runtime Environment:
# SIGSEGV (0xb) at pc=0xb7b62856, pid=24497, tid=3084424896
# JRE version: 6.0_15-b03
# Java VM: Java HotSpot(TM) Client VM (14.1-b02 mixed mode, sharing linux-x86 )
# Problematic frame:
# C [libpango -1.0.so.0+0x23856] pango_layout_new+0x36
# An error report file with more information is saved as:
# /home/user/he_err_pid24497.log

users would probably report that MOSKitt crashed when they were modeling but not the error message from the console. Further, if maintenance engineers would work on a Windows machine, they would not be able to reproduce the error.

- **Error Handling**
  When the error can be replicated, maintenance engineers debug in order to understand what leads to the error and which part of the code is erroneous. Then they change the code in order to avoid the crash to happen again. In our scenario, the debugging would reveal that the problem is caused by the libpango library and the maintenance engineer would be required to debug the library.

**TO-BE** In the following, we describe how the FastFix system handles the application crash in this scenario. There are several stages to FastFix’s contribution to this type of applications: pre-deployment interventions, execution monitoring and post-execution fault handling.

1. **Monitor application execution**
   In the pre-deployment phase, MOSKitt’s code will be instrumented by injecting additional byte code instructions in the application byte code. During execution, sensors collect information about *user interactions, application execution events* and the current *context*, especially sources of non-determinism like *user input* or *calls to external components* are observed and their effects are stored.

2. **Send generated, rich error report to FastFix server**
   If the application crashes, an *error report* is generated and sent to the FastFix server automatically. This error report will contain a timestamped sequence of all context and runtime monitoring information gathered during MOSKitt’s execution. The error report will only be sent with the user’s consent and all user input information will be anonymized on the client side.

3. **Open ticket**
   On the server side, the *error report* will be stored and a *ticket* will be opened to enable the management of the issue.

4. **Replay of erroneous application execution**
   The execution of MOSKitt that lead to the crash can be replayed stepwise on several levels of abstraction. During *replay*, the maintenance engineer can pause
the replay and inspect the current state of MOSKitt represented by information on variable values, context information and user interaction history. By doing so, the maintenance engineer can search for the cause of the error similar to debugging but without the need to reproduce exactly the same environment as the users’ because all user input and sources of nondeterminism were recorded in advance. In our scenario, the maintenance engineer could observe that directly before the application crashes, the libpango library has been called which gives a hint that this library might cause the error.

5. **Recommend similar error reports**

Additionally, the FastFix system presents references to similar, related error reports to the engineer. These references can be used to detect if the current error is a duplicate and as hints where to look for solutions that can be transferred to the current problem.

6. **Deploy new application model to self-heal**

To provide a solution to the problem a new application model that prohibits the application state of the crash to be reached in application execution is generated and deployed to the FastFix client.

### 2.2.3.3 Configuration Error – Main Strategy: Solution Recommendation or Self-Healing

**Error Description**  
Application configuration denotes the practice to set flexible settings of an application instance like GUI language, license files or database properties at startup of the application. Configuration properties are handed over to an application as a configuration file, as application parameters, as items in the registry or they are hardcoded. If the configuration is done in a wrong way, this can lead to problems as worse as causing the application to crash.

In this scenario we examine configuration errors in TXT Production Planning, a production planning software that is part of the TXTPerform suite of the FastFix consortium member TXT e-solutions.

TXT Production Planning uses Microsoft technology (MS SQL Server 2008, Excel, Visual Basic 6, C++) as implementation platform. As TXT is using a closed source business model, we will assume in this scenario that the FastFix system does not have access to the source code of TXT Production Planning and consequently has to accomplish its functionality without. TXT Production planning can be used in a client-server version and a standalone Desktop version.

**Symptoms**  
In the concrete scenario we are examining here, TXT Production Planning shows the error message depicted in Figure 2.2 immediately after startup and closes after the user acknowledges the error message.
Several possible causes may be responsible for the occurrence of this symptom, namely the database properties are wrong (e.g. typo in the IP address of the database server), the database credentials have changed or the SQL server has not been started.

The following items describe the state of the art procedure to handle configuration errors.

- **Error Reproduction**
  In order to reproduce the error, maintenance engineers need to know the database credentials the user has used in this attempt to connect to the database. As they are usually not included in the error report, the maintenance engineers have to contact the user to get to know them. Then they either know immediately that there is an error, e.g. because the credentials are wrong or have been changed, or they use the credentials and try themselves to connect to the database.

- **Error Handling**
  In order to fix the configuration problem, maintenance engineers either contact the user directly to set up the correct configuration (e.g. by a remote login or telephone support) or they provide the user with a tutorial showing how to correctly set up the applications configuration.

In the following, we describe how the FastFix system handles configuration errors.

1. **Extract configuration information**
   Information about the configuration of the application is extracted as part of the application’s context by special sensors.

2. **Observe occurrence of error**
   FastFix sensors detect the occurrence of a configuration error.

3. **Send rich, generated error report**
   After detection of the configuration error, an error report is generated automatically that contains information about the configuration. This error report is sent to the FastFix server automatically.
4. **Open ticket**
   On arrival of the *error report* at the FastFix server, it is stored and a *ticket* is created automatically allowing to manage the issue.

5. **Display configuration information**
   As the *error report* contains information about the application’s *configuration*, maintenance engineers can examine the current configuration directly, without the need of contacting the user, when they are working on the ticket.

6. **Recommend correct configuration**
   Based on its *knowledge base* (that was populated when the same error has occurred on another application instance before), the FastFix system recognizes the configuration problem and suggests a solution consisting of the correct configuration.

7. **Self-heal by applying correct configuration**
   The correct configuration is - with the approval of the maintenance engineer - sent to the FastFix client and is ensured by applying a patch to the application. The advantage of this approach is that the user does not have to take any action to update his current configuration to the correct one. This last step is optional. The maintenance engineer can also decide to use the solution suggested in the previous step, test it and change the configuration on the user’s machine in another way.

### 2.2.3.4 Usability Problem on Mobile Device – Main Strategy: Fault Replication and Solution Recommendation

**Error Description**  
Usability is the “ease with which a user can learn to operate, prepare inputs for and interpret outputs of a system” [2] and is an important property of a software system. This is especially the case for current mobile devices whose user interface consists of a small screen forcing user interface designers to distribute information on several views and provide navigation opportunities between them. Usability problems are hard to detect because they involve human interpretations and cannot be measured clearly.

In this scenario we focus on a fictive e-mail client that runs on a mobile device such as an iPhone or Android phone and is implemented in Objective C or Java. We assume that the source code of the app is available and can be used within the FastFix system.

**Symptoms**  
The user of the e-mail client app is confused about how to achieve a certain task like replying to an e-mail. Symptoms that indicate this problem are switching between views randomly or no interaction with important controls and views. An example for random view switching is the view sequence “Start application” -> “View email” -> “Select email” -> “Settings” -> “Select Email” -> “Settings” -> “Select email” -> “Settings” -> “Select Email” -> “Reply to email” -> “Select email” -> “Reply to email” -> “Reply to email”.


Cause  Several causes can be responsible for the symptoms described above, namely if the user has no experience with the mobile device or the app or if the user interface of the app is designed badly.

AS-IS  The following items describe the state of the art procedure to handle usability problems.

- Error Reproduction
  For the maintenance engineers it is not possible to exactly reproduce the usability error as they are different persons and have different experience levels, e.g. they probably are more familiar with the mobile device or the target app. In order to detect the usability problem the only way for them is to observe users when interacting with the target application. This requires them to be at the same location as the user or doing a screen capture session of the user’s mobile device.

- Error Handling
  If a usability error is detected either by user observation or by a user reporting it, maintenance engineers redesign the user interface of the mobile application, implement the new design, test it and deploy it using a traditional update channel.

TO-BE  In the following, we describe how the FastFix system handles usability problems.

1. Monitor app execution
   The FastFix sensors monitor the inputs of the user and the execution of the app. Especially, text entered by the user, views visited, buttons pressed and gestures performed are detected and recorded. This data is aggregated into sessions and transmitted to the FastFix server. Before transmission, confidential data is obfuscated.

2. Compare navigation paths and create user profile
   On the FastFix server, usage data from several users arrives and the navigation path consisting of visited views of different users can be compared. The comparison can be used to detect that a user is a beginner and to identify important views by counting the frequencies of visits among all users.

3. Replay user sessions
   As the execution of the app has been recorded, it can be replayed and maintenance engineers can observe how users use and navigate the app.

4. Detect usability problem
   If important views are learned from usage data or indicated manually, usability problems of a user can be detected by inspecting user sessions and checking if important views are visited within the session. If not, the system concludes that there might be a usability problem.

5. Recommend solution
   If the detected usability problem is user specific, e.g. because the user is a beginner, the FastFix system can push a message to the user containing a link to a tutorial or
video educating the user how to use the application. If several users face the same
problem, the user interface may have been designed in a confusing way. The Fast-
Fix system can compute recommendations about how to change the link structure
between views based on navigation paths or how to change the size of certain user
interface elements.

2.3 Functional Requirements

2.3.1 User Requirements

One of the main goals of FastFix is to improve the way maintenance engineers work
and to support them with appropriate information and tools. Hence, in this section
we summarize the functional requirements necessary to support maintenance engineers in
their daily work. These requirements were posed by the maintenance engineers themselves
during the interviews. We focus on general requirements for the FastFix system. For
requirements of specific FastFix components see [5], [9], [3] and [4].

2.3.1.1 General Maintenance Environment Design

This section summarizes some principles that will guide the development of the mainte-
nance environment in general.

- Graphical Representation of Information. Information should be represented in a
graphical way to provide easier overview for maintenance engineers. Examples for representing
information graphically is to use swim lanes to represent parallel execution of several threads,
to use a timeline to represent the sequence of commit logs, to graphically highlight differences in code
between two different versions of a file or to use a graph to visualize dependencies between packages.

- Summarization of Data. Statistical analysis should be used to aggregate information
and provide useful statements.

- Different Levels of Granularity. Information should be presented in different levels
of granularity and it should be possible for the maintenance engineer to choose the
level of granularity based on his current information needs. For example, system
engineers will need a relative low level of information granularity for application
related context information. The FastFix system should provide different granularity level hierarchies for applica-
tion and user context. The application context information should be representable
in the following levels of granularity: Component, Plugin, Package, Class, Method,
Function, External Call.

- Personalization. The FastFix system should provide a customizable maintenance
environment on the server side based on the personal expertise and preferences of
maintenance engineers.
• Filtering. The possibility to filter information and view only relevant information for the current context should be provided.

• Information Navigability. It should be possible to easily navigate between different kinds of related information. An example for such a navigability would be that a click on a method in the stack trace of an error would show the source code of that method.

2.3.1.2 Error Reproduction and Replay

The following requirements account for a more comfortable and efficient error reproduction and replay, i.e. executing an application step by step until an error occurs.

• Provide Error Reproduction Information. In cases where possible, information about how a specific error can be reproduced should be provided by the FastFix system. An example for such information is a step-by-step guide like “First, start the application. Then enter ‘foo’ into text field ‘Name’ and press Button ‘Submit’ - the application will crash”.

• Enable Video-like Error Replay. If an error is connected with a graphical user interface, a way to replay the execution of the application step by step and see what happened on the GUI should be provided. In such a case, the maintenance engineer sees the same screen as the user has seen.

• Context Augmented Debugging. As the FastFix system collects lots of context information during the execution of an application, this information will be used to augment the information a debugger provides.

• Unified View of Debugging Information. Static (application version, OS version, ...) and dynamic (CPU load, running threads) context information should be added to the information that is provided by a classic debugger (stack trace, variable values) and all information should be shown in a unified way during debugging.

• Highlighting of Changing Information. Information that changed between the last and the current debugging step should be determined and highlighted.

• Context-aware Breakpoints. The possibility to set context-aware breakpoints that are triggered when a context variable is assigned a certain value or range should be provided during context-aware debugging should be provided. For example, to set a breakpoint triggering if the CPU usage increases over 80 %.

2.3.1.3 Information Recommendation

In order to help the maintenance engineer to deal with the huge amount of information that is available and to provide him with information that is relevant for his current task and in his current situation, the FastFix system will use a recommendation based approach. In this section an overview of recommendation tasks within FastFix is given.

• Recommend Similar Bugs. Previous bugs that are similar to the current one should be recommended to maintenance engineers. They can use these in order to check if the current bug is a duplicate of them or get inspiration on possible solutions if the solution of previous bugs is recorded.
• Recommend Possible Cause For Symptoms. In order to remove a problem from a software system, the first step is to find the cause of it while only symptoms can be observed directly. Consequently, the FastFix system should provide a mechanism to detect, store and query relationships between symptoms and causes of problems.

• Recommend Possible Solutions For Causes. After identifying the cause of a problem, the cause has to be removed. Similar to the relationship between symptoms and causes, a mechanism to represent, store and query relationships between causes and possible solutions should be provided.

• Recommend Instructions For Complex Patches. If a patch is complex to apply, e.g. because it requires many steps to be carried out, the FastFix system should present a step by step instruction to the maintenance engineer about what to do and in which sequence. This allows the maintenance engineer to go along such an instruction and helps to conduct all necessary steps during the application of a patch in the correct order.

2.3.1.4 User Control in Self-Healing

• User Control over Deployment. When a patch for an error is generated automatically and ready to be applied it should not be applied automatically but the maintenance engineer should be given control over the application of the patch. This gives the maintenance engineer control over patch deployment and allows him to inspect and test the patch before deployment. Further, a mechanism to rollback patches should also be provided.

• Testing of Patches. The FastFix system should provide the possibility to test generated patches, especially automatically generated ones.

2.3.1.5 Relevant Information To Be Covered

Maintenance engineers need different types of information to perform their tasks. In this section, we summarize the information sources and types of information that are considered within FastFix.

• Provide Logs. The FastFix system should provide information comprised in logs in a single point of access.

• Modification History of Artifacts. A modification history should be provided by the FastFix system for different types of artifacts.

• Reports From Users. The FastFix system should provide a facility to collect, store and present problem reports from users.

• Support for Finding Experts For Certain Issues. In order to enable a maintenance engineer to get access to information that was not externalized and that is consequently not represented in a system, experts on certain issues have to be identified. In order to acquire information the maintenance engineer can get in contact with an expert. An example for an expert on source code is the author of the source code or a person that has used and modified a piece of code several times.
2.3.2 System Requirements

This section comprises the fundamental functional requirements for the FastFix system that are necessary to provide the functionality that is described in the scenarios above. This section focuses on general requirements for the FastFix system and for certain FastFix components. For requirements of specific FastFix components see also [5], [9], [3] and [4].

2.3.2.1 Context Elicitation and User Profiling

In order for the FastFix system to be able to identify execution error symptoms, performance degradation or changes in user behavior, the system should collect context information while the host application is running and while the user interacts with it. This context information has to comprise data about the host application itself, about the user as well as about system related data. The collected data should represent information in different levels of granularity. Data collection has to be independent from the application and its environment.

- Capture Context Selectively. Capturing context implicitly includes selecting the relevant context sources - through enabling and disabling the appropriate sensors - as well as associating context with the corresponding pieces of knowledge when necessary.

- Configurability of Monitoring. The monitoring should be configurable in the sense that the level of detail of monitoring can be adjusted to e.g. collect more detailed information if an attack is anticipated and collect less detailed information if the danger is over and to create less overhead workload.

- Context Processing. Context should be used in order to produce higher-level context information through projection, aggregation, filtering, etc.

- Store Context. The elicited context should be persistently stored in order to enable reuse of experience.

- Query Context. The context system should provide two context-querying services:
  - Push: in this case the knowledge or the user query will be annotated with context without any other component explicitly asking for this information
  - Pull: in this case the component explicitly asks for the context of a specific knowledge and the context system delivers the context in a predefined interaction language.

- Observe Change. Change is one important event type that should be triggered. The context system should observe which changes occur to which properties of which entities.

- Interpret Captured Information. The interpretation process includes generalizing the context, identifying relevant context, identifying/classifying context, and associating context to other knowledge. A special case for the interpretation of the captured information is the prediction of context.
User Profiling

- Log Interaction. All information about a user’s behavior should be captured in a log file. Examples of such interactions are for example: “the user posts a query” or “the user performs a search”.

- Discover Preference. The preferences of the user should be discovered while analyzing previous user interactions and personal context.

- Synchronize Context. The context system should determine the start and the end of a user’s session (the so-called context sessionization). A user session can be for example a set of actions that are related to a particular use case of the target application.

- Discover Usage Patterns. The context system should discover usage patterns in order to identify some important characteristics of the user. Therefore, log information should be analyzed from different perspectives and for different periods of time (i.e. set of events). For this analysis, machine learning and data mining mechanisms should be applied. However the interpretation strategy should be extensible to further mechanisms.

2.3.2.2 Event Correlation

Event correlation determines how the gathered information is going to be used to draw conclusions about the kind of problems the application is facing and how they can be removed.

- Correlation Rules Knowledge Base. To solve a problem that occurs in an application, the problem itself first needs to be recognized. The FastFix system should use event correlation techniques to draw conclusions about possible problems. To that end, it has to create and maintain a knowledge base covering problems, causes, symptoms, solutions and the relationships between them (e.g. the reason for problem X can be cause Y).

- Correlation Engine Requirements. The knowledge base will be used to determine problems, causes, and solutions based on a probabilistic model. To provide re-usable functionality, certain event correlation techniques have to be realized as a service facade to the knowledge base. The correlation engine shall provide the following functionality:
  - Detect patterns of events that often precede or cause an error
  - Determine likely problems for given symptoms
  - Retrieve probable causes for given problems
  - Suggest possible solutions to given problems and their causes
  - Anticipate failures
  - Detect performance degradation trends

In order to obtain the relationships described above, mining techniques should be used to detect such relationships from collected data.
- Pattern Matching Module. In order to prevent failures and performance degradation, the correlation engine should use pattern matching on the monitored events. The pattern matching module should be specialized in the early identification of event patterns that indicate failure.

2.3.2.3 Fault Replication

The fault replication related requirements determine the functionality that allows replicating errors as close to the real circumstances as possible.

- Deterministic Replication of Execution. In order to analyze application faults its execution has to be deterministically replayed in the FastFix server. Replaying application execution aims to understand what lead to faults and provides the basis for the work of maintenance engineers. To enable automatic execution replaying, the FastFix system has to utilize context information gathered by the context elicitation components. Further, the replay functionality has to provide facilities to be used interactively by maintenance engineers.

- Execution Recording. As a prerequisite for the replication of application execution, data about the execution has to be collected and transferred to the FastFix server. The recording mechanism has to integrate with the components performing context elicitation and user profiling. Several non-functional requirements strongly influence the execution recording functionality, as described in Section 2.4.

- Advanced Data Obfuscation. In order to provide privacy, collected data has to be obfuscated. However, data obfuscation techniques may corrupt execution traces. The FastFix execution recorder has to use advanced data obfuscation techniques that overcome these problems.

2.3.2.4 Patch Generation and Self-Healing

- Selection of Appropriate Patches. Once a problem is identified, the FastFix system shall select appropriate patches from the patch database that can correct the erroneous behavior of the application.

- Generation of a New Patch. If no existing patch solves the given problem, the FastFix system shall be able to synthesize new patches from model structures and control behaviors.

- Application of Patch at Runtime. Self-healing can be defined as patch generation and application of the generated patch at runtime. This requires control over the application at runtime.

2.4 Non Functional Requirements

The non functional requirements for the FastFix system are summarized in this section. This section focuses on general requirements for the FastFix system. For requirements of specific FastFix components see [5], [9], [3] and [4].
2.4.1 General Applicability
The FastFix system has to be potentially applicable to all types of applications. More specifically, the architecture and conceptual models have to be designed in an extensible way to enable the portability of the FastFix system to other types of applications that are not considered in the FastFix prototype implementation. Porting and using the FastFix system to another type of application may require substantial efforts but it should be possible without starting from scratch.

2.4.2 Performance
User interactions and application context have to be monitored efficiently by creating only as much additional workload as required. The context system has to perform its activities mainly in the background, in most of the cases without a concrete interaction with the user. The user should not face severe performance problems while using the target application due to monitoring.

Further, in case a maintenance engineer actively pulls a certain type of information within the FastFix server maintenance environment, the information has to be acquired and presented within a reasonable time span.

2.4.3 Extensibility of Context System
To deal with heterogeneous environments and changes in technical surroundings, the context system has to be extensible. This extensibility ensures that future technologies and interactions can be supported by writing a sensor and integrating it into the overall system. An extension to address a new technology or interaction shall not require modification of the existing context system and it has to be possible to add it easily to the overall system.

2.4.4 Privacy
As the FastFix system will monitor, process and interpret user input, measures have to be taken to ensure the privacy of user data. Sensitive data like user names or passwords have to be obfuscated without limiting FastFix functionality and consequently research is needed about how both goals can be accomplished. Special care has to be taken when transmitting user data from FastFix client to FastFix server and when storing user input on both client and server.

2.5 Mockups
In this section we illustrate how the FastFix system could look like in the form of user interface mockups. The mockups shall give a first simple impression of the supported complex functionality and the presentation of the results.

2.5.1 Error Report Generation
Figure 2.3 shows a mockup of the error reporting user interface of the FastFix system. The FastFix system provides application independent user interfaces to enable error reporting
even for applications that do not include this feature. The error report shown in the
mockup also contains an automatically generated list of steps to reproduce the error,
with sensitive data being obfuscated.

![Error Reporting User Interface](image)

**Steps to reproduce:**

1. Open MOSKitt
2. Import data file (file data obfuscated)
3. Click button named “btn_Transform”

![FastFix Error Reporting](image)

Figure 2.3: Error Reporting User Interface (Mockup).

On the maintenance site automatically generated error reports arrive in the issue tracker
system. Figure 2.4 illustrates an automatically generated error report denoting an unhan-
dled exception in the MOSKitt application (cf. Section 2.2.3.2). We used the issue tracker
Trac\(^{4}\) to demonstrate the functionality. However, the FastFix system will be extensible
to work with any issue tracking system.

\(^{4}\)http://trac.edgewall.org
2.5.2 Context Augmented Fault Replication

In order to analyze application faults its execution has to be deterministically replayed on the maintenance site. Replaying application execution aims at understanding what led to faults and provides the basis for the work of maintenance engineers.
Figure 2.5: Context Augmented Fault Replication (Mockup).

Figure 2.5 shows a mockup of the fault replay functionality of the FastFix system integrated into the Eclipse IDE. The replay functionality provides facilities to be used interactively by maintenance engineers. To this end, engineers may move forward and backward within a video-like reproduction of the application execution. While watching the user interactions on the GUI the maintenance engineer can observe the user and application context in a graphical or textual representation and in different levels of granularity.

3http://www.eclipse.org
Figure 2.6: Context Augmented Fault Replication - System Context (Mockup).

Figure 2.6 illustrates how the maintenance engineer may observe the system context of the application while actually going through the execution as it happened on the user’s machine.
D2.3: User Requirements and Conceptual Architecture

Figure 2.7: Context Augmented Debugging (Mockup).

Figure 2.7 shows a mockup of context augmented debugging. Here, the debugging facilities of the maintenance engineer’s IDE are augmented with several context features. First, context-aware breakpoints allow to control the debugging flow depending on context information in addition to conventional information like variable ranges. Second, engineers can observe context information like conventional code variables. Further, the source code view can be enriched with context information to provide additional knowledge e.g. in context menus.
2.5.3 Error Cause Recommendation

![Mockup of FastFix system recommending similar error reports and error cause recommendations.]

Figure 2.8: Error Cause and Error Report Recommendation (Mockup).

The mockup in Figure 2.8 shows how the FastFix system might recommend similar error reports to the maintenance engineer based on similarity metrics and event correlation. This information will be included in extensions of issue tracker systems. The FastFix system can also recommend possible error causes based on the reported symptoms including an according probability measure. Further, links to relevant parts in the source code as well as the responsible developer may be recommended.
2.5.4 Self-Healing and User Feedback

Figure 2.9 illustrates how the FastFix system might support maintenance engineers in reusing or generating control objectives that will be used to self-heal target applications (for detailed information on self-healing concepts see [4]). The system behavior can be represented graphically. Further, control objectives may be designed graphically for better understanding.
Your reported issue with the application **MOSKitt** has been successfully investigated. The error is now resolved.

Thank you for using a FastFix compliant application!

---

**Figure 2.10**: User Feedback Interface (Mockup).

Finally, Figure 2.10 shows a mockup of a FastFix user feedback interface. The FastFix user feedback interfaces enable target applications to provide feedback to the user regarding the maintenance of the application, even if the application itself does not include user interfaces for this purpose.
3 Architecture

This section describes the overall conceptual architecture of the FastFix system. It contains a brief overview of the system’s main components as well as a description of their most important public interfaces.

The described architecture and components provide the basis for the realization of the user and system requirements illustrated before.

3.1 FastFix System Overview

To facilitate remote software maintenance the FastFix system provides a set of tools that range from automatic error report generation to self-healing mechanisms. To this end, the FastFix system continuously monitors target applications and client environments gathering context information on application execution and user interaction. The FastFix system then uses correlation techniques to detect incorrect execution patterns and changes in user behavior. Once a problem is identified, the FastFix system provides several strategies to resolve it. In the maintenance center, faults are replicated while the collected context data can provide additional information on error causes. Further, an event correlation system identifies causes and solutions to given errors. Finally, the FastFix system creates patches for detected causes and deploys them to the target application.

The design of the FastFix context aware system is partly based on the work by Maalej [8]. The FastFix system comprises two different platforms, the FastFix client platform running on the system where the target application is executing, and the FastFix server platform that runs on the maintenance site and is connected to all clients over a network. Both platforms comprise several specialized components that interoperate to realize the FastFix functionalities described in Section 2.2.2 In the following we summarize these components.
Figure 3.1: FastFix Client Platform Overview.

Figure 3.1 shows an overview of the FastFix client platform. We distinguish between six main components. The client data store (Section 3.2.1) serves as a data access layer to the FastFix client ontologies. The context system (Section 3.2.2) provides context elicitation and processing functionality. The client self-healing system (Section 3.2.3) is in charge of applying patches to the target applications. The application bridge (Section 3.2.5) bundles all interfaces to communicate with the target applications in a simple and extensible way. The error reporting and user feedback system (Section 3.2.4) comprises user interfaces to collect user input for error reports and to provide feedback to the user. The communication system (Section 3.2.6) finally is in charge of handling data exchange between the client and the server platform.
Figure 3.2: FastFix Server Platform Overview.

Figure 3.2 shows an overview of the FastFix server platform. We distinguish between seven main components. The server data store (Section 3.3.1) serves as a data access layer to the FastFix server ontologies. The event correlation system (Section 3.3.2) draws conclusions about the kind of problems the target application is facing and what possible causes are. The fault replication system (Section 3.3.3) provides means to deterministically replay the application execution. The server self-healing component (Section 3.3.4) creates patches for the observed fault. The maintenance engineering UI system (Section 3.3.5) provides FastFix functionality to the maintenance engineers in the form of specialized user interfaces that include issue related information and error replay facilities. The maintenance environment bridge (Section 3.3.6) bundles all interfaces to communicate with differently equipped maintenance environments in a simple and extensible way. Finally, the communication system handles the data exchange between server and client platform as described before.

3.2 FastFix Client Platform API

3.2.1 Client Data Store

This component implements Semantic Web technologies and standards. It manages metadata and URIs and provides facilities to access metadata and the FastFix ontologies. Figure 3.3 shows its main interfaces.

The FastFix client ontologies will be a subset of the ontologies described in Section 4.2 focusing on concepts and relationships needed to represent the information that is sensed by the sensors of the context system.
3.2.2 Context System

The FastFix context system includes functionality for context elicitation and context processing [5]. The context system consists of components that provide means to register events (Context Hub), define context collection interfaces (Sensing), and facilitate information processing (Processing). Figure 3.4 shows the most important interfaces of these components. The context system model is described in detail in [3].

![Figure 3.4: Context System Interfaces.]

3.2.3 Client Self-Healing System

The client self-healing system provides interfaces concerning patches and their application to the target application. Figure 3.5 shows its main interfaces. In the FastFix system, a generated patch is applied through the control of a supervisor. A patch describes permitted application behaviors and the supervisor component enforces that the application
execution does not deviate from the allowed behavior. The supervisor observes context events originating from the applications execution and updates its knowledge regarding the current execution and application state. A supervisor applies the “patched” application behaviors by preventing some method executions so that the application behavior does not deviate from the ones provided by the patch. For more details on the functionality and concepts of the self-healing system see [1].

The method executions that can be prevented by the supervisor correspond to controllable events. These events are defined when the FastFix client is installed and are automatically deployed into the target application (by modifying its bytecode for instance).

![Client Self-Healing System Interfaces](image)

Figure 3.5: Client Self-Healing System Interfaces.

### 3.2.4 Error Reporting and User Feedback System

Figure 3.6 shows the main interfaces of the error reporting and user feedback system. This component provides two sets of functionality. First, it comprises user interfaces to collect user input for error reports and to provide feedback to the user. Second, it includes classes that perform necessary pre-processing tasks to allow for fault replication on the maintenance site (see [3] for further details).

The error reporting service may request error reports from the user, while the error report generator actually generates error reports based on context information. The generation of error reports requires also the processing of the application trace after the execution. Post-execution trace processing includes important tasks like data obfuscation.
3.2.5 Application Bridge

The application bridge provides interfaces to the target applications for sensing, patch deployment, reporting, and feedback. The component detaches the FastFix system from concrete target applications and bundles application access in a single component. The application bridge contains implementations of context system interfaces and client self-healing system interfaces that are target-application specific.
D2.3: User Requirements and Conceptual Architecture

![Figure 3.7: Application Bridge Interfaces.](image)

As shown in Figure 3.7, the ApplicationBridgeService facade provides facilities to register and manage new target applications in the FastFix system. Once a target application is registered, the facade allows to install context sensors, to patch the target application, to request error reports, and to show FastFix feedback to the target application user.

Below we further describe how the application bridge interacts with the context, client self-healing, and error reporting and user feedback systems.

### 3.2.5.1 Context System Dependencies

The context system defines context collection interfaces in the Sensing component. These interfaces define how sensors have to be structured to work within the FastFix system. Concrete implementations of sensors naturally depend strongly on the target application [5]. The application bridge detaches the context system from concrete target applications by providing a layer for concrete implementations of its interfaces.

As shown in Figure 3.7, the application bridge allows to install sensors for a specific target application. To this end, concrete and tailored implementations of the context system’s SensingStrategy interface can be registered for a specific target application. Once installed, these sensors may then directly provide context events to the context system.
3.2.5.2 Client Self-Healing System Dependencies

The client self-healing system controls how the FastFix system applies patches to target applications, and how these patches may be rolled back. The application bridge is in charge of removing the dependency between the self-healing concepts defined in the client self-healing system and concrete target applications. To this end, the application bridge installs actuators in the specific target application and actually applies a given patch. It further takes care of restarting the target application if needed, keeps track of problems during the patching process, and rolls back patches if needed.

3.2.5.3 Error Reporting and User Feedback System Dependencies

The error reporting and user feedback system comprises facilities to collect user input for error reports and to provide feedback to the user. It includes interfaces that define error reports and user feedback and further provides components that request input from and show information to end users of the target application. The latter components allow the FastFix system to interact with the user even if the target application does not provide means for this interaction.

Future FastFix compliant application, however, may implement error reporting and user feedback interfaces themselves. The application bridge takes care of the target application capabilities by forwarding error report requests either to the FastFix error reporting system or to the target application itself.

3.2.6 Communication System

The communication system adds a transparency layer on the network connection between the FastFix client and server platforms. It includes functionality to exchange data between the FastFix client platform and the FastFix server platform.

3.3 FastFix Server Platform API

3.3.1 Server Data Store

The server data store provides similar functionality and interfaces as the client data store described in Section 3.2.1.

The FastFix server ontologies encompass the FastFix client ontologies to enable the FastFix server to interpret the data that is received from the FastFix client and that are encoded using the FastFix client ontologies. Additionally, the FastFix server ontologies encompass the knowledge base about causes, symptoms, issues and solutions and their relationships and maintenance specific information. All of the ontologies described in Section 4.2 are included in the FastFix server ontologies.

3.3.2 Event Correlation System

This component draws conclusions about the kind of problems the target application is facing and what possible causes are. As shown in Figure 3.8 the main components within the event correlation system include the event correlator, the pattern matcher, and an
according correlation knowledge base. The event correlator uses the pattern matcher to draw conclusions.

The event aggregation subsystem is in charge of aggregating and combining the monitoring data of different users of the target application. It will keep track of grouping similar errors reported by different users. The component includes a statistical analyzer which provides means to evaluate frequency and content of error reports. Further, a conflict resolver component includes strategies to handle contradictory reports. For more details on the functionality and concepts of the event correlation system see [9].

![Event Correlation System Interfaces](image)

Figure 3.8: Event Correlation System Interfaces.

### 3.3.3 Fault Replication System

The fault replication system provides means to deterministically replay the application execution. Its main interfaces are shown in Figure 3.9. First, the fault replication system provides information required to reproduce errors, such as error reports and steps to reproduce the error. It also uses the event correlation system to gather reports for similar errors. Second, the fault replication system provides objects used to replay errors. For more details on the functionality and concepts of the fault replication system see [3].
3.3.4 Server Self-Healing System

The server Self-Healing system is in charge of selecting and computing patches. A patch describes the permitted application behavior and the supervisor component enforces that the application execution does not deviate from the allowed behavior. The main interfaces are shown in Figure 3.9. It is mainly composed of a patch generator that allows for automatic patch creation as well as a control objective designer that allows for automatic design of a model of the behavior not leading to reported errors. Before any patch can be generated, a model of the target application behavior must be generated from its source code. This is done by the model extractor and is only performed once for a given application version.

When an error is reported, a control objective is first designed. It represents a formal description of application behaviors that do not lead to the error. This control objective can be manually designed by expertise. Alternatively, it may be automatically designed from the conclusions drawn by the event correlation system.

This control objective is then used in order to generate a patch, with respect to the model of the target application behaviors and the set of deployed controllable events (i.e. methods whose execution can be prevented as defined in the client self-healing system).

When a patch is generated, it is then validated through the patch validator. Such a validation typically consists of running tests on the patched version of the target application. Patch validation can also be used for patch selection whenever several patches could be applied, in order to determine the most suitable one. For more details on the functionality and concepts of the self-healing component see [I].
3.3.5 Maintenance Engineering UI System

The maintenance engineering UI system presents FastFix functionality to the maintenance engineers in the form of user interfaces that include issue related information, error replay facilities, as well as means to interact with the self-healing system. As shown in Figure 3.11, it comprises a user interface to deliver error related information like relevant code portions to maintenance engineers. Further, it contains a user interface for error replay and a user interface that visualizes control objectives.

The maintenance engineering UI system defines interfaces that will be specialized for different maintenance environment applications like specific issue trackers and integrated development environments.

Figure 3.11: Maintenance Engineering UI System Interfaces.

3.3.6 Maintenance Environment Bridge

The maintenance environment bridge bundles all interfaces to communicate with differently equipped maintenance environments in a simple and extensible way. It comprises interfaces to components within the maintenance environment, such as the Eclipse debugger and issue trackers like Bugzilla and Trac.
As shown in Figure 3.12, the maintenance environment bridge provides interfaces to add issues in issue trackers. Further it encapsulates access to the source code of an application. To this end, it contains abstract adapter interfaces to bug tracker and versioning systems. Finally, the component provides an interface to replay errors in the maintenance environments.

In the following we further describe how the maintenance environment bridge interacts with the maintenance engineering UI, fault replication, event correlation, server self-healing, and server data store systems.

### 3.3.6.1 Event Correlation System Dependencies

The interaction between the maintenance environment bridge and the event correlation system is twofold. First, the maintenance environment bridge requests probable solutions for a given issue from the event correlation system. Second, the event correlation system uses the maintenance environment bridge to access the issue tracker system at the maintenance site. This way, new tickets can be entered regardless what issue tracker is present in the maintenance environment.
3.3.6.2 Fault Replication System Dependencies

The fault replication system provides information needed to replay errors in development environments. To replay an error, a maintenance engineer uses the according FastFix feature in the debugger, realized as an extension (e.g. an additional view) that interacts with the maintenance environment bridge. The maintenance environment bridge then requests objects needed for the error replay from the fault replication system.

Further, the maintenance environment bridge queries the fault replication system for getting similar error reports to a given issue, when this is requested by the maintenance engineering UI system.

3.3.6.3 Server Self-Healing System Dependencies

The maintenance environment bridge provides means to manually specify a control objective (i.e. property to be ensured, from which a patch is generated) or to visualize a control objective that has been automatically designed from the conclusion drawn by the event correlation system.

3.3.6.4 Maintenance Engineering UI System Dependencies

The error information UI component uses the maintenance environment bridge to show additional information about an error, such as similar errors and relevant code parts. In order to replay an error, the error replay UI component requests a replay object from the maintenance environment bridge that contains control and context information for error replay.

3.3.6.5 Server Data Store Dependencies

The maintenance environment bridge uses the server data store to manage issues. Since issues have to be represented both within the FastFix system and within the issue tracker system used by the maintenance engineer, we introduce the concept of an TrackerIssue. This allows the representation of issues within multiple issue tracker systems. The issue manager component in the server data store is in charge of mapping between FastFix internal and issue tracker specific issues.
4 Ontologies and Conceptual Model

Ontologies are used within the FastFix system as data representation mechanism. In this chapter we motivate our decision of using ontologies (4.1), describe ontologies that cover important domains (4.2), and give examples about how ontologies are used within the FastFix system (4.3 and 4.4).

Data in FastFix is mainly created by sensors and processed by components like Event-Correlation, FaultReplication and SelfHealing. These data have to be represented and stored. Alternatives for data representation are relational databases, XML files and ontologies. FastFix will use ontologies, specifically OWL ontologies, as data representation technique because they posses some advantages over the other options. These advantages are described in section 4.3.

All FastFix ontologies together play the role of a data schema which defines data that will be stored in the FastFix system. Consequently, all types of data that should be stored and processed in FastFix components have to be modeled as concepts in one of the ontologies.

The ontologies can be viewed as a machine-interpretable instantiation of the FastFix conceptual models. Consequently, the FastFix system becomes aware of its application domain described by the conceptual model, can reason about it and behave intelligently, e.g. by adopting to specific situations or making appropriate recommendations.

Reasoning using the ontologies described in this section as a basis can be performed in two ways. First, a Description Logics based reasoner can be used to reason about the ontologies without any additional efforts. Such a reasoner can reason about taxonomy relationships or classification of instances by mapping them against concept definitions in the ontologies or can simply be used as querying interface to query information store in an ontology. Second, user defined inference procedures can be written that use the ontologies as input data and built on DL reasoning capabilities. Such user defined inference procedures will be written if a FastFix component has a reasoning need that cannot be handled with standard DL reasoning. Mainly, this will take place in the Event Correlation Component of FastFix (for more details on how ontologies will be used within the Event Correlation Component see Section 4.4 or 49).

4.1 Reasons For Using Ontologies

There are several reasons for the FastFix system to use ontologies as data and knowledge representation mechanism and we shortly summarize them in this section.

- **Flexibility** The “schema” (in DL language TBox) of an ontology can be changed and expanded easily. This fits well into the FastFix situation where it is only partly known which context information will be used and where additional sensors may be added during

---

1Some reasons are inspired by a tutorial of Noy and McGuinness available at http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html
the project. More specific, if a new sensor sensing a new attribute should be added to the FastFix system, the new attribute can be added to the local ontology and used on the local FastFix client. When data from local FastFix client is merged with ontology data from other clients, the new attribute is automatically taken over into the merged ontology without the need of additional efforts.

**Support of Distributed Data** When merging data from different sources, which is the case in the FastFix system employing several client nodes and a server node, consistency checking is supported by reasoners. This consistency checking mechanism can be used to find inconsistencies between data from different sources.

**Reasoning Support** Ontologies grew out of the Semantic web effort to make data machine-interpretable in contrast to e.g. text. Therefore, data encoded using ontologies can be interpreted and reasoned about by machines.

Reasoning support for ontologies is enabled by well-defined semantics and the use of Description Logics based reasoners. This enables querying ontology based data and inference of statements that are not directly asserted but implied by the asserted ones. Examples of “built-in” inference capabilities are consistency check of ontologies (especially important when several ontologies are merged), subsumption calculation and query-like retrieval of all individuals of a concept, among others. Domain-specific inference schemes can be implemented based on domain rules and realized in form of user-defined procedures that built on OWL reasoning mechanisms.

Examples for such domain specific inference schemes within FastFix are the application of event correlation rules to classify the current situation (represented by context events captured by FastFix sensors) to be an instance of e.g. a security attack.

**Semantic Interoperability** A major advantage of ontologies is that they cover semantics of the data they are representing. This enables semantic interoperability instead of syntactic interoperability. For example, if data denoting the same concept are stored using different terms both data items would be treated as different concepts by a pure syntactic system. A semantic system, in contrast, could figure out that both data items refer to the same concept and use this knowledge when doing reasoning or querying. Prerequisite for this capability is that both terms are defined to be synonyms in an ontology.

A specific example used within the FastFix system are the two terms *Bug* and *Fault* which we defined as being synonyms in our ontology. If one subsystem uses the term *Bug* to represent its data and another *Fault*, in systems supporting syntactic interoperability *Bug* and *Fault* would be judged to be different concepts, which is wrong. Because of the semantic interoperability, more specifically the assertion that *Bug* and *Fault* are the same concepts, a query retrieving all *Faults* would return *Bugs* also.

Another example is comparing syntactically different, but semantically equivalent user inputs.

**Explicit Domain Assumptions and Common Understanding** Using ontologies to capture domain knowledge results in a explicit, formal description of domain concepts and relationships. Domain assumptions have to be explicitly stated in order to enable
the reasoning mechanism to consider them, resulting in their faster and easier detection, comprehension and change of them.

As a consequence, common understanding is enabled and promoted among the Fast-Fix consortium members. In this scenario, the ontologies representing domain models are used as basis for discussion among consortium members about important concepts. Results of such discussions are represented as ontologies again and this process is done iteratively. As a result of the process, a common understanding and definition of concepts and relationships is reached which is important as in many cases components implemented by different partners have to cooperate in order to fulfill FastFix requirements. This is exactly one of the main purposes of this section on ontologies.

**Reuse of Knowledge** Information represented as an ontology can be modularized into different ontologies. If an ontology refers to concepts defined in another ontology, the ontology import mechanism can be used. This architecture facilitates reuse of information in ontologies in several different contexts. For example, the maintenance ontologies developed within FastFix can be used in other projects or by other people. Usage in this context means to just use them as they are or to refine and expand them.

### 4.2 Description of FastFix Ontologies

#### 4.2.1 Overview

In order to provide the desired functionality described in the requirements, FastFix components need to acquire, process and store information about several domains. Because this information is stored in form of ontologies, the ontologies have to cover all of these domains. In order to reduce complexity and ease reusability, a modularized approach has been taken and several ontologies have been developed. Each ontology covers one domain that is of interest for FastFix. The ontologies themselves may be split into parts, too. Figure [1] gives an overview of the FastFix ontologies.

- **Application Execution Ontology** covers information about the runtime behavior of the target application like runtime environment, application structure, and threading information.
- **Domain Ontology** includes information about the application domain of the target application. Exemplary application domains are supply chain management (logistics, delivery, invoices) or Mapping Domain (map, location, distance, ...).
- **User Interaction Ontology** covers information about the current user and its interaction with the target application. Examples for covered information are interaction history, interaction taxonomy, experience level, and preferences of a user.
- **Maintenance Ontology** covers information that is relevant for software maintenance like maintenance activities, error taxonomy and a cause-issue-symptom knowledge base.

This section concentrates on concepts and relationships that are general and that are of interest within several FastFix components. Further details of conceptual models of different FastFix components can be found in [5], [9], [3] and [4].
4.2.2 Application Execution Ontology

The Application Execution Ontology represents information about the runtime behavior of the target application like runtime environment, application structure, and threading information. Having available this information enables to examine the current state of the application at a specific point in time and reason about the static (e.g. dependencies or bad influences between different application components or between application and environment) and dynamic properties (e.g. occurring event patterns indicating problems) of the application.

It is divided into several parts each representing a fraction of the application execution information. The parts are described in this section in more detail. All concepts described in this section are represented in the ontology ApplicationExecutionOntology.owl.

Execution Context  This part represents the context in which an application execution takes place. This includes information about hardware and software without the target application itself.

A distinction is made between StaticContextInformation, information that does not change during the execution of the target application, and DynamicContextInformation, information that changes during the execution. StaticContextInformation subsumes information about the current operating system and the runtime environment like JVM. DynamicContextInformation consists of a sequence of events occurring in the context of the target application like NetworkEvent or OSEvent but not events within the target application itself. The sequence of EnvironmentEvents is ordered using the event time stamp as ordering criterion. All ContextInformation together represents the current context of the target application.
Event Taxonomy  As the main unit of dynamic information within the FastFix system are events that are sensed and created by sensors, the Application Execution Ontology contains an event taxonomy which is depicted in Figure 4.3 and Figure 4.4. ContextEvents are segmented into three major types of events: UserEvent denoting events originating from the user of the target application (described in Section 4.2.4), ApplicationEvent representing all events taking place within the target application and EnvironmentEvent constituting of all events outside the target application (both described in the current Section). Each ContextEvent has a StartTime and an EndTime and events that have the same start and end time are called Atomic.

Figure 4.2: Definition of application execution context

Figure 4.3: EnvironmentEvent Taxonomy
**Application Structure**  This part defines constructs representing the structure of the target application and is shown in Figure 4.5. The structure in Figure 4.5 is the structure which is typical for an OSGi application written in Java. But it can be adopted or extended for other types of applications. Each `ApplicationPart` corresponds to a level of abstraction at which the target application can be inspected and analyzed, e.g. when looking for dependencies this can be done on class or package level. Dependencies are represented using `dependsOn` relationship on `ApplicationPart`. This enables to model different dependencies like bundle depends on bundle or class depends on another class. Further, the version attribute of `ApplicationPart` indicates that all levels from `Application` down to `Statement` have a version associated.
Application Execution Model  Figure 4.6 shows the Application Execution Model which represents information about runtime behavior within the target application. The Application under consideration consists of one or more Threads representing program execution. A Thread executes Methods of the Application and accesses Attributes of Classes and Variables of Methods (an access can be a read or write access as shown in the Event Taxonomy in Figure 4.4). Further, a Thread makes calls to ExternalComponents like Library or OperatingSystem.

In order to gain insight into the application at runtime, the target application is instrumented. The instrumentation is represented by relation hasInstrumentation and a type of Instrumentation which can be on source code level, byte code level or system level. Alternatively, there is no runtime instrumentation but log files are analyzed, which is denoted by InternalLoggingInstrumentation (because the log files are produced by an instrumentation mechanism that is internal to the target application and independent of FastFix instrumentation).
Configuration  An interesting property of a runtime instance of an application is its configuration. A simple configuration scheme is depicted in Figure 4.5. The configuration consists of a set of ConfigurationParameters and each ConfigurationParameter has a ConfigurationValue associated.

4.2.3 Domain Ontology

The term “application domain” denotes the environment in which an application is used. For example, the application domain of an SCM application is the area of supply chain management containing concepts like supplier, product, logistics or invoice.

An important part of maintenance is understanding the application domain logic encoded in the application and some errors, especially configuration and functionality errors, are strongly related to the application domain. Hence, it is useful to represent concepts of the application domain in an ontology to enable reasoning about them. For example, reasoning can check if errors occur frequently during execution of a certain application domain task or if the application configuration is conflicting with relationships in the application domain (e.g. a certain configuration parameter is set to a physical unit that is not possible within the application domain). Figure 4.7 shows a small, incomplete formalization of the SCM application domain. All concepts denoted in the figure are represented in the OWL file ScmDomainOntology.owl.
4.2.4 User Interaction Ontology

The User Interaction Ontology covers information about the current user and its interaction with the target application. Examples for covered information are interaction history, interaction taxonomy, experience level, and preferences of a user. The ontology can be used as basis to reason about the user like detecting normal and abnormal usage patterns, determining the user’s experience or preferences.

All concepts described in this section are represented in the ontology UserInteractionOntology.owl.

**User Event Taxonomy**  The user event taxonomy shown in Figure 4.8 gives an overview over events that can originate from a user’s interaction with the target application.
**User Interaction and User Profile**  The User Interaction Ontology represents information about the interaction of a user with the target application and about the user itself. Figure 4.9 illustrates the user-application domain. In the figure the main components of the User Interaction Ontology are represented. A user has capabilities, preferences, knowledge, and intentions. As mentioned in [5], we will focus on modeling and inferring *Capabilities* and *Preferences* of users because we believe that this is the most valuable information for the event correlator. User preferences and capabilities can be inferred from the age, duration, and frequency attributes of the events performed using statistical analysis and frequent pattern mining. More details about determining user preferences and capabilities can be found in [5].

*UserEvents*, described in more detail in the taxonomy above, and the responses to these events from target application and environment, namely *ApplicationEvents* and *EnvironmentEvents*, can be organized into sessions. A *Session* consists of an ordered set of related events that occurred in a specific period of time. The analysis of the events of a session through machine learning techniques, such as sequence pattern matching, can lead to the inference of additional user capabilities and preferences.

![User Interaction Domain diagram](image)

**Figure 4.9: User Interaction Domain**

### 4.2.5 Maintenance Ontology

The maintenance ontology represents information about the maintenance process and important concepts therein. An important part is the knowledge base representing causes, issues and symptoms that can be used to store and reason about the relationships between them and e.g. retrieve possible causes and probable solutions for a problem. The maintenance ontology further enables the FastFix server component to reason about the maintenance domain (e.g. similar bugs) and to adapt to the current context of the maintenance engineer (e.g. showing certain information only if the maintenance engineer performs an activity in which the information is useful).

All concepts described in this section are represented in the ontology *MaintenanceOntology.owl*.

**Conceptual Error Model** Important maintenance tasks include the reproduction of errors, finding the cause of an error and finally fixing the error. We call the according in-
formation needed in activities like these “conceptual error model” and describe this model in the remainder of this section.

Figure 4.10 shows an overview of the Conceptual Error Model containing three major concepts Cause, Issue and Symptom.

*Cause* represents reasons for the appearance of certain *Issues* within the system. Examples for *Causes* are *Bugs* or high CPU usage. A specific *Cause* can give rise to several *Issues* and an *Issue* can have several *Causes*. An important subtype of *Cause* is *Fault* which is defined by [2] as “a design or coding mistake that may cause abnormal component behavior”.

*Issue* represents a certain *Problem* that exists within the target application like *PerformanceDegradation* or *ExceptionThrown*. *Issue* and *Problem* are synonyms. An important subtype of *Issue* is *Error* or *ErroneousState*, defined by [2] as “a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure”.

*Symptoms* are all forms in which an *Issue* manifests and can be observed. A specific *Issue* can manifest in several *Symptoms* and a specific *Symptom* can potentially be the manifestation of several *Issues*. For example, a possible *Symptom* for the *Issue PerformanceDegradation* is high waiting time, and *ApplicationCrash* is a *Symptom* for the *Issue ExceptionThrown* (there can be situations in which an unhandled exception does not lead to an application crash). Some symptoms can be observed by the end user and we call them *UserPerceivableSymptom*. A synonym of *Symptom is Failure* which is defined by [2] as “a deviation between the specification and the actual behavior”. In the FastFix system, *Symptoms* are detected by *Sensors* and are represented as *ContextEvents*.

![Diagram of Conceptual Error Model](image)

Figure 4.10: Overview of Conceptual Error Model

To formalize different kinds of *Causes* further, a Cause Taxonomy has been developed which is shown in Figure 4.11. It shows that there are mainly two types of *Causes* for *Issues*, namely *ImplementationCause* and *ExternalCause*. The first represents implementation mistakes like *Bugs* while the second represents *Causes* originating from outside the target application like *UserInteractionCause* (e.g. abnormal or malign user behavior) or *OSCause* (e.g. high CPU load caused by other applications).
**Fix Related Concepts**  Removing bugs and mitigating symptoms are important tasks of a maintenance engineer. In this paragraph we describe important concepts related to fixing. An overview can be found in Figure 4.12.

Central concept is a *FixAction*, i.e. an action carried out by a maintenance engineer with the goal to address an *Issue*. A *FixAction* can be a *PreventiveAction*, representing “a modification after delivery to detect and correct latent faults before they become operational faults” (definition taken from [7]), or a *CorrectiveAction*, representing “a reactive modification performed after delivery to correct discovered problems” (definition from [7]). A *FixAction* applies a *Patch* to the software system. A Patch is defined as “any modification to a source or object program” (definition taken from [H]) and can be a *CodeChange*, a *ConfigurationChange* or a change of the supervisor model. A *Patch* either removes a *Cause* completely from the application (all *Issues* and *Symptoms* originating by that *Cause* are removed), then it is called a *Solution*, or it removes or mitigates a *Symptom* only (e.g. by restarting the application). Then it is called a *Workaround* or *FaultMitigation*. *Solutions* are preferable to *Workarounds* but if the *Cause* is not known or it is time intensive to develop and apply a *Solution*, a *Workaround* is often used in practice.

![Figure 4.11: Cause Taxonomy](image)
**Error Report Model**  An *ErrorReport* encompasses all data that have been gathered by the FastFix sensors and that are related to an *Error* or *Issue*. Figure 4.13 gives an overview. Because the data from an *ErrorReport* can be used to replay a specific *Error*, an *ErrorReport* represents a *ReplayableError* which is a subclass of *Error*. The *ErrorReport* consists of sequences of *ContextEvents*, *AttributeAccesses* of Threads and *MethodCalls*. *ContextInformation* is also contained in an *ErrorReport*.

**Maintenance Process**  The activities a maintenance engineer conducts constitute the maintenance process. In order to be able to adapt to the current context and task of the maintenance engineer, knowledge about the maintenance process is formalized in the Maintenance Ontology. Figure 4.14 shows important concepts in this domain.

A *MaintenanceProcess* consists of *MaintenanceActivities* which have preceding activities (represented via *previousActivity*) and succeeding activities (represented by *followingActivity*). A taxonomy following Harjani et al. [6] of *MaintenanceActivities* is also given. It lists several kinds of *MaintenanceActivities* like *ProblemUnderstanding* or *Implementation*. 
4.3 Examples of Ontology Usage

In this section we give concrete examples of how the ontologies described above can be used in the four FastFix scenarios and in the FastFix system.

4.3.1 General

Within the FastFix system, all kinds of data are stored in form of ontologies. In particular, all events created by sensors are encoded by instantiating concepts that have been defined previously in an ontology.

An important part of the FastFix event correlator is a knowledge base representing relationships between causes, issues, symptoms, solutions and workarounds. This knowledge base will be implemented using ontologies by representing causes, issues, symptoms, solutions and workarounds as concepts and the relationships between them as properties. In order to e.g. find possible causes C for a specific symptom S the ontology can be queried and all causes for which a relation to S is asserted or can be deduced are returned. Following the representation of causes, issues and symptoms in the Maintenance Ontology above, the relationships between symptoms and causes is indirect via issues and hence a property chain reasoning mechanism is used which reasons that if symptom S is a symptom of issue I and issue I is caused by cause C then C is a possible cause for S.

4.3.2 Input Validation Scenario

Ontologies are used to store information that is required in the Input Validation scenario described in Section 2.4.3.1. In particular, patches and workarounds for each issue I
are represented in ontologies, the relationship between a patch and the related issue is modeled using a property and the ontology can consequently be queried to retrieve patches or workarounds that can be used to deal with a specific issue.

4.3.3 Crashing Java Application Scenario

Scenario 2 (described in Section 2.2.3.2) depends heavily on information stored in ontologies. The concepts of causes, issues, symptoms and solutions and the relationships between them will be represented in ontology. More specifically, there will be a taxonomy for each of those terms in the ontology that can be instantiated. This kind of representation allows to retrieve probable causes for observed symptoms and solutions for specific causes by reasoning along asserted relationships and collecting information. Further, error reports and the fact that two error reports are similar can be represented and stored in ontologies enabling to retrieve similar error reports for a specific error report ER.

4.3.4 Configuration Scenario

Section 2.2.3.3 describes a scenario focusing on errors occurring in the configuration of an application. In order to be able to do so, a definition of the current configuration and correctness axioms have to be defined first. This can be done in ontologies. Further, DL subsumption reasoning may be used to deduce if the correct configuration subsumes the current configuration and hence the current configuration can be judged to be correct.

4.3.5 Mobile Scenario

The Mobile Scenario described in Section 2.2.3.4 deals with errors connected to 3rd party components, environment issues and usability. As essential information for this scenario, information about the current system configuration, installed 3rd party components, the environment configuration and information about the current user (like preferences and experience) is represented and stored using ontologies. As in the Crashing Java Application Scenario, information about causes, symptom, issues and solutions together with the relationships between them is stored in ontologies. The assertion of this knowledge enables querying for probable causes for a certain symptom and retrieve possible solutions for specific issues by reasoning along asserted relationships and collecting the desired information.

4.4 Usage of Ontologies in FastFix Components

In this section we briefly summarize how ontologies will be used in each FastFix component.

4.4.1 Context System

- Information store for event and user information
  The information about events and the user acquired by sensor will be stored in an

2Applicability of this approach has still to be researched and evaluated
ontology. Examples of such information are the type of event, start and end time and the user initiating the event.

- **Retrieval of information**
  As all information about events will be stored in ontologies, these can be queried in order to retrieve event information. In the query process, ontology reasoning is used in order to not only retrieve asserted facts but also inferred ones. Examples for possible queries are to retrieve all UserEvents of user X (using taxonomy reasoning to determine which events are UserEvents) or all events that started in a certain period of time.

- **Context processing and aggregation**
  Context processing and aggregation like e.g. substituting a set of KeyPressed events by a Typing event will be done using ontologies as data basis. Context processing and aggregation will be implemented partly by using ontology reasoning mechanisms and partly by user defined procedures.

- **Ontology data as input for user profiler**
  The user profiler will use the information in the ontologies to detect preferences and levels of expertise. More specifically, the event taxonomy and User Interaction Ontology will be used to reason about a specific user like judging the user to be an expert if he frequently uses keyboard shortcuts. The user profiler will be implemented partly by ontology reasoning mechanisms and partly by user defined procedures.

### 4.4.2 Event Correlation System

- **Information store and information retrieval**
  All domain data for the event correlation system will be stored in form of ontologies, giving a way to represent taxonomies, equivalence and relationships of any type between concepts in the conceptual error model.

- **Reasoning about taxonomies**
  Event correlation will filter ontological data before using it with the help of ontological taxonomy reasoning, e.g. consider only certain types of events. Further, rules will be formulated on abstract concepts and applied to events asserted on a specific level because the ontological reasoning on the event taxonomy closes the gap and matches specific events to general events.

- **Reasoning about categorization**
  Ontological categorization reasoning allows to design context specific rules that e.g. are only applicable for concrete types of applications or events. Ontological reasoning will be used to categorize the current context and match it to a context definition. For example, a concept UserEvent can be defined as “all events initiated by a user” in the ontology and a DL reasoner automatically classifies all instances of Event that have a relationship with an instance of a user along the property initiatedBy as being an UserEvent without the need to assert for each of those instances that they are a UserEvent. A rule can be defined that is only applicable to this concept UserEvent.
• Reasoning using an ontological knowledge base
  In order to find probable causes for symptoms or recommend possible solutions, the Event Correlation component will use a knowledge base that is encoded in form of ontologies. If the knowledge store provides information about certain issues, associating them with symptoms, causes and solutions, event correlation can use this information to provide these items associated with a concrete issue. In addition to that, causality is also a relationship between instances that can be detected within the correlation process, hence this kind of relationship can be represented and queried during root-cause analysis.

• Semantic interoperability
  Ontologies will be used to provide semantic interoperability in reasoning. For example, in the ontology the fact that the terms Bug and Fault are synonyms can be represented. DL reasoning will consider this equivalence relationship and queries to retrieve all instances of Bug will contain instances of Fault also.

4.4.3 Fault Replication System

• Information store and information retrieval
  All input data for the fault replication system will be stored in form of ontologies. The fault replication system will, during execution reply on the server side, recover context information from the ontologies. Examples of such information is the current application version, process creation sequences and user input data.

• Vocabulary for error reports
  Concepts defined in FastFix ontologies will also be used to describe error reports and their replayable versions.

4.4.4 Client Self-Healing System

• Selection of Actuators
  The functionality of the client self-healing system mainly consists of applying a patch and deploying actuators (i.e. control points) in the system. Ontologies can provide an elegant way to select or describe the actuators. For example, if we want to control "all the methods that are triggered through user interaction" - such as buttons, fields, mouse focus, etc. - this can be described using ontologies.

4.4.5 Server Self-Healing System

• Information store and information retrieval
  Context information necessary to compute or adapt a patch to the current context can be queried by the server self-healing system from the ontologies.

• Input to compute control objective
  Sequences of events, encoded as ontology events and detected by the event correlation system, are used in order to compute a control objective of intended or unintended application behavior.
• Ontological patch repository
  Existing patches for known issues are stored in ontologies and can be retrieved by querying the ontologies.
5 Intelligence in FastFix

In the FastFix requirements described above phrases like “detect event patterns”, “find probable causes for problems” or “recommend possible solutions” appear. Such requirements need techniques and methods from the fields of artificial intelligence (AI) or computational intelligence (CI) in order to be accomplished. This section shows how such technologies can be aligned with FastFix requirements and how different technologies can be combined. We consider three different technologies here, namely ontologies (e.g. OWL), rule-base systems (e.g. Drools, Prolog) and machine learning algorithms (e.g. frequent pattern mining, statistics, ...). This section attempts to give a rough overview of possible usage of technologies and does not aim at defining clearly which technology will be used within the FastFix system. This will be an active question of research and we will try different possibilities during the duration of the FastFix project to find out which technology fits best to the requirements.

5.1 Alignment of Intelligence Technologies to FastFix Requirements

The FastFix event correlation component is one of the crucial parts of the FastFix system and plays an important role in order to fulfill the FastFix requirements. It is important to align the AI or CI technology used within the event correlation component with the FastFix requirements, i.e. to make sure that the AI or CI technology offers enough functionality to match the requirements.

In order to get a first overview, the following table summarizes the FastFix requirements, organized into the four FastFix scenarios (rows) and general ones, that are relevant for the AI or CI component and tries to give a first, rough judgement how well each requirement can be fulfilled by each AI or CI technology (columns). The table summarizes our current judgment based on our experience and our research efforts in this issue. We will continue to research and experiment in order to discover which AI technology can support which requirement and which advantages, disadvantages and constraints exist for each AI or CI technology.

As this is only a rough judgement, we use “Not possible” to indicate that the requirement cannot be fulfilled by a technology, “Open” to indicate that further research is needed to establish if a requirement can be fulfilled by a technology and “Possible” to indicate that a requirement can be fulfilled by a technology.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Validation Scenario</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Ontologies</td>
<td>Rule-based systems</td>
<td>Machine learning</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>------------</td>
<td>---------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Recognize SQL statement in textual user input</td>
<td>Open</td>
<td>Possible (Text matching rules)</td>
<td>Possible (Text mining)</td>
</tr>
<tr>
<td>Detect occurrence of defined event sequence</td>
<td>Open</td>
<td>Possible</td>
<td>Open</td>
</tr>
<tr>
<td>Retrieve patch for current issue from knowledge base</td>
<td>Possible</td>
<td>Open</td>
<td>Possible (Learning of mapping from report keywords to patch)</td>
</tr>
<tr>
<td>Retrieve workaround for current issue if no patch exists</td>
<td>Possible</td>
<td>Open</td>
<td>Possible (Learning of mapping from report keywords to patch)</td>
</tr>
<tr>
<td>Retrieve similar error reports</td>
<td>Possible</td>
<td>Not possible</td>
<td>Possible (Textual similarity calculation)</td>
</tr>
<tr>
<td>Find probable causes for symptoms (from knowledge base)</td>
<td>Possible</td>
<td>Possible (Condition: Relationships represented)</td>
<td>Possible (Learning of mapping from report keywords to cause)</td>
</tr>
<tr>
<td>Detect relation between causes and symptoms</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Frequency statistics)</td>
</tr>
<tr>
<td>Retrieve solutions for causes (from knowledge base)</td>
<td>Possible</td>
<td>Possible (Condition: Relationships represented)</td>
<td>Possible (Learning of mapping from report keywords to solution)</td>
</tr>
</tbody>
</table>

**Java Application Crash Scenario**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect event patterns occurring frequently</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Frequent pattern mining)</td>
</tr>
<tr>
<td>Detect relationship between issue and event pattern from data</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Frequency statistics)</td>
</tr>
</tbody>
</table>

**Rule-based systems**

- Text matching rules
- Text mining

**Machine learning**

- Learning of mapping from report keywords to patch
- Learning of mapping from report keywords to cause
- Learning of mapping from report keywords to solution
### Requirement

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect relation between solution and cause from data</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Frequency statistics)</td>
</tr>
<tr>
<td>Detect event patterns that lead to a crash</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Frequent pattern mining)</td>
</tr>
</tbody>
</table>

### Configuration Scenario

<table>
<thead>
<tr>
<th>Configuration Scenario</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect defined, malign application configuration</td>
<td>Possible (Condition: malign configuration has to be defined)</td>
<td>Possible (Condition: malign configuration has to be defined)</td>
<td>Open</td>
</tr>
<tr>
<td>Recommend configuration fix (i.e. change of configuration that resolves the issue)</td>
<td>Possible (Condition: Relationship represented)</td>
<td>Not possible</td>
<td>Open</td>
</tr>
</tbody>
</table>

### Usability Error Scenario

<table>
<thead>
<tr>
<th>Usability Error Scenario</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect usability issues</td>
<td>Open</td>
<td>Possible (Condition: Issue represented)</td>
<td>Possible (Comparison of different navigational paths)</td>
</tr>
<tr>
<td>Discover usage patterns</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible (Navigational path mining)</td>
</tr>
<tr>
<td>Discover environment incompatibilities</td>
<td>Possible (Condition: Incompatibilities represented)</td>
<td>Possible (Condition: Incompatibilities represented)</td>
<td>Open</td>
</tr>
</tbody>
</table>

### General

<table>
<thead>
<tr>
<th>General</th>
<th>Ontologies</th>
<th>Rule-based systems</th>
<th>Machine learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Storage</td>
<td>Possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
<tr>
<td>Store data that are collected by the sensors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertainty</td>
<td>Open</td>
<td>Open</td>
<td>Possible</td>
</tr>
<tr>
<td>Represent non-binary facts and relationships (e.g. probable causes and how probable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Represent knowledge base with information about causes, issues, symptoms, solutions and workarounds</td>
<td>Possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>
5.2 Possible Combinations of Intelligence Technologies

This section shortly describes ideas and approaches about how different AI or CI technologies can be combined within the FastFix system.

Rule-based Systems and Machine Learning

Rule Mining  Rule mining denotes the use of a machine learning algorithm to detect patterns in data and represent these patterns in the form of rules. This approach can be transferred to event correlation by learning temporal patterns of events using a frequent pattern mining algorithm and coding the results of this algorithm in form of rules.

As event correlation is one of the main functionalities of the FastFix system and a rule-based system will be one of the main components of the FastFix event correlation engine, this approach will be used within the FastFix system heavily.

Selection of next rule to fire  A problem in rule-based systems is conflict resolution which denotes the problem to choose the next rule to fire when the antecedent of several rules is satisfied and consequently all of them could be fired. Machine learning algorithms can be used to determine the next rule to fire within the rule engine.

As this approach would mean to modify the core of a rule engine this will not be likely to be pursued within the FastFix system.

Rule-based Systems and Ontologies

Incorporating rules in ontologies  Rules can capture application logic that cannot be represented by a DL-based ontology alone. In order to expand the expressiveness of ontologies and be able to capture such application logic, rules can be embedded into ontologies. Those rules can be defined based on concepts defined in the ontology and a reasoner supporting (DL-safe) rules can consider them in his reasoning procedure. An example for such an approach is SWRL.

As all data within the FastFix system will be represented in the form of ontologies and it is easy to incorporate rules in ontologies this approach will likely be used in the FastFix system.

Call DL reasoning from rules  Ontologies provide convenient ways to represent domain knowledge and reason about it. In order to reuse these capabilities of ontologies, ontology reasoning can be incorporated in rule based systems to e.g. classify variables that occur in a rules antecedent. An example of such an approach is the rule-based system Drools with a custom operator interfacing to Pellet for DL reasoning or Prolog with additional rules implementing OWL reasoning.

As one of the main components of the FastFix event correlator will be a rule-based system this approach will be used in the FastFix system.

Machine Learning and Ontologies

Using ontologies as data source for ML algorithms  Data stored in form of ontologies can be used as input data for machine learning algorithms either in their original form or
after data transformation into another format.

As all data in the FastFix system will be stored in form of ontologies this will be the main mechanism to feed data into machine learning algorithms within the FastFix system.

**Using ML to populate ontologies** If textual descriptions of the domain under consideration are available, special text mining algorithms can be used to extract domain knowledge from the text and represent it in the form of ontologies.

This type of combination will not be investigated within FastFix.
6 Summary

In this deliverable, we gave an overview of FastFix and presented a blueprint of the FastFix system. We summarized the requirements for FastFix originating from interviews and analysis of desired FastFix functionality, sketched a high level architecture of the future FastFix system, described ontologies that serve as information representation mechanism within FastFix and discussed how artificial intelligence techniques can be used to accomplish the goals of FastFix.

In the requirements part, we identified five error types that FastFix will potentially have to deal with, namely application crash, configuration error, dependency error, hardware or resource error and input validation error.

Further we described ten possible strategies to deal with a concrete error depending on its complexity and the current context.

These strategies are error report recommendation, error report generation, fault replication, context augmented fault replication, error cause recommendation, error cause explanation, solution recommendation, fix recommendation, patch generation, and self-healing.

In order to link error types and error strategies to real world situations, we described four scenarios that deal with errors occurring in software applications of the FastFix industry partners. These scenarios named input validation scenario, Java application crash scenario, configuration scenario, and usability scenario will focus the implementation of the FastFix system.

The main user requirements, posed by maintenance engineers in interviews, are an advanced presentation of information (supporting different levels of granularity, the possibility to filter information, and a graphical and unified view of information from different sources), the provision of information for error reproduction, the recommendation of context relevant information (similar bugs, possible causes, and probable solutions), and maintaining control over the deployment of patches. Information sources that were ranked relevant were logging information, modification history of artifacts, and user reports.

The main system requirements which are necessary to provide the desired functionality for FastFix are to observe changes of context variables, to selectively capture context information, to discover preferences and experience of users, to maintain a knowledge base about important entities and their relationships (more specifically of causes, issues, solutions, and workarounds), to mine frequent usage patterns, to detect performance degradation problems, to replicate a fault deterministically, to record the execution of an application, to generate a patch for a problem, and to self-heal (i.e. automatically repair) an application.

Non functional requirements for the FastFix system include adaptability to all kinds of applications, to employ lightweight monitoring in order to not increase the resource needs of applications considerably, extensibility, and privacy to protect private data by using data obfuscation techniques.
Within the architecture part, we sketched the structure of the FastFix system consisting of a client and a server platform. The *FastFix client platform* consists of six main components, namely error reporting and user feedback system, application bridge, context system, client self-healing system, client data store and communication system. Seven main components constitute the *FastFix server platform:* Maintenance engineering UI, maintenance environment bridge, fault replication system, event correlation system, server self-healing system, server data store and communication system.

In the ontology part we described the data representation mechanism of FastFix and its high level conceptual model. The *application execution ontology* covers information about execution context, event taxonomy, application structure, application configuration, and application execution. The *domain ontology* represents concepts from the application domain of the target application and the *user interaction ontology* covers user events, the profile of a user, and the interaction between a user and the target application. Finally, the *maintenance ontology* represents the conceptual error model, a cause taxonomy, concepts related to patches, and bug fixing activities, an error report model and a model of the maintenance process and its activities.

In the last part we described how several AI techniques – namely ontologies, rule based systems, and machine learning – can be used to realize the requirements that depend on reasoning and intelligence. We further discussed shortly how these AI techniques can be combined to achieve a maximum benefit of them within FastFix.
Bibliography


