ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

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


Lead contractor: Lero
Author(s): Benoit Gaudin, Mike Hinchey, Paddy Nixon, Raian Ali, Newres Al Haider.
Submission date: November 2011
Dissemination level: Restricted

Project co-founded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document reports on a new version of the FastFix D6.3 deliverable, i.e. implementation of the 1st prototype of the FastFix self-healing and patch generation components. The initial goal of this deliverable has been modified in order to allow earlier implementation of the self-healing, hence an earlier evaluation of both the FastFIX self-healing and patch generation approaches. First this document recalls in Section 2 the main objectives and expected functionalities of these components, that will be developed as part of the FastFix project. Three main features of the self-healing components (model extraction, patch generation and runtime supervision) and their requirements are presented. Section 3 describes the current implementation of these features, corresponding to the delivery of this 1st prototype. It mainly focuses on the model extraction aspect on which the other ones are relying on. The extracted models are Finite State Machines (FSM). Section 3.2 describes the structure of these components and the constructs and specificities of the Java programming languages that it takes into account in order to extract Finite State Machines. The functionalities and usage of the implemented features are illustrated with an example in Section 4.1. Finally, Section 4 provides scalability results and describes the next steps to be performed from the implementation detailed in Section 3 in order to achieve the objectives described in Section 2.
## Contents

1 Introduction 4

2 Component Objectives 5
   2.1 Conceptual Models for Self-Healing and Patch Generation 5
   2.2 Requirements for the Self-Healing Component 7
   2.3 Automatic Patch Generation Component 8
   2.4 Existing Tools and Libraries 9
   2.5 Summary 10

3 FastFixSH: 1st Prototype Iteration 11
   3.1 Technology and Libraries Used 11
   3.2 Self-Healing Component 12
      3.2.1 Model Extraction 12
      3.2.2 Supervisor Deployment 15
      3.2.3 Runtime Monitoring and Supervisor (Patch) Application 18
   3.3 Automatic Patch Generation Component 19
   3.4 Summary and Limitations 20

4 Results and Next Steps 22
   4.1 Illustrative Example 22
   4.2 Scalability Results 26
   4.3 Next Steps 30

Bibliography 31
1 Introduction

This document reports on the implementation of the 1st prototype of the self-healing and FastFix patch generation components, FastFixSH (for FastFix Self-Healing). Unlike its initial version, this deliverable not only considers the 1st prototype of the patch generation component, but also the 1st prototype of the self-healing component. Therefore this deliverable presents limited patch generation capabilities but more developed self-healing features at this stage of the project. Such a modification will make it possible earlier evaluation of the FastFix self-healing and patch generation approach. New versions of deliverables D6.4 and D6.5 have also been considered and detailed in an updated version of the FastFix Description of Work.

This report describes the current features of the prototype and the efforts provided, with respect to the functionalities to be achieved according to the conceptual model for FastFix self-healing and patch generation. This document first recalls in Section 2 the main objectives and expected functionalities of these components, that will be developed as part of the FastFix project. The contents of Section 2 recalls the FastFix conceptual models for self-healing introduced in [7]. More specifically, Section 2.1 recalls the three main features of the self-healing component: model extraction, patch generation and runtime supervision. Sections 2.2 and 2.3 respectively introduce requirements for the self-healing and patch generation components. Section 3 describes the current implementation of these features, corresponding to the delivery of this 1st prototype, and driven by the requirements described in Sections 2.2 and 2.3. It mainly focuses on the model extraction aspects on which the other ones rely on. Section 3.2 describes the structure of these components and the constructs and specificities of the Java programming languages that it takes into account in order to extract Finite State Machines. The functionalities and usage of the implemented features are illustrated with an example in Section 4.1. Finally, Section 4 provides results and describes the next steps to be performed from the implementation detailed in Section 3 in order to achieve the objectives described in Section 2. These results provide insights regarding the scalability of the model extraction and patch generation components. For this, it has been applied to three open source projects. Two of these projects possess more than 7000 method declarations. Scalability results regarding the runtime supervision mechanism require further development and improvement of the current implementation. Therefore, such results are not provided in this deliverable.
2 Component Objectives

This section aims to position the current implementation of the FastFix self-healing and patch generation components with respect to the conceptual models described in the corresponding Conceptual Models deliverable (D6.2, [7]). First Section 2.1 recalls the conceptual models introduced in [7], then Sections 2.2 and 2.3 respectively provide requirements that drive our implementation for the self-healing and patch generation component prototypes.

2.1 Conceptual Models for Self-Healing and Patch Generation

In this section, we recall notions about the conceptual models for self-healing and patch generation in FastFix. It is based on the corresponding deliverable: [7]. The FastFix platform follows the client-server architecture. Self-healing is involved in both the client and the server side. The server side is in charge of automatically extracting models from the application code as well as generating patches. The client side is in charge of supervision deployment and runtime execution. The 1st iteration of the self-healing and patch generation components do not follow this architecture, which will be achieved in a further version. However, it takes it into consideration by implementing the different features independently.

Figure 2.1 gives an overview of the conceptual models for self-healing and patch generation in FastFix. The corresponding approach for self-healing can be split into three phases. The first phase occurs before the system is deployed and consists of extracting models of the application behaviors and deploying sensors with supervision capabilities into the application.

The second phase of the approach occurs at runtime, where the application is under supervision. This supervision mechanism is based on deployed sensors and the previously extracted model of the system behaviors. At first this model of the system is used as a supervisor model, i.e. a patch, that has no impact on the implemented behaviors.

The third phase corresponds to patch generation and occurs whenever an issue is detected in the application. The information sensed on the application is used to first generate a control objective, i.e. a pattern characterizing traces that lead to the observed issue. Such patterns are determined by the Event Correlation Component. In some cases, it can be achieved automatically. In other cases, this process may involve expertise and is performed manually or semi-automatically. From such patterns and the model of the application behaviors computed before deployment, the patch generator component computes new models of the behaviors of the system. These new models correspond to
patched behaviors. When the application is relaunched, the new patch is taken into account by the sensors and actuators and they can observe and act on the system in order to prevent executions leading to the previously observed issue.

The 3 phases for self-healing are represented in Figure 2.1 with the circles numbered 1, 2 and 3. Phase 1' corresponds to sensors and actuators deployment, which is performed through the Context System component and Phase 2' corresponds to the determination of a control objective, which is provided by the Event Correlation Component. In order to implement self-healing in FastFix, the self-healing component itself must implement:

1. a model extraction component.
2. a runtime supervision mechanism.
3. a patch generator component.

Section 2.2 details the features that the model extraction component must achieve as well as the mechanism for runtime supervision of the application. Section 2.3 details features that must be implemented by the patch generator component.

![Figure 2.1: A Software Control Approach to Self-Healing.](image)

The FastFix platform follows a client-server architecture. The self-healing component spreads over both the client and the server side. The patch generation component, which is part of the self-healing component is located on the server side, as well as the model extraction component. The client part of the self-healing mainly deals with the runtime supervision. This component is embedded into sensors which also contain models of patches and apply them using actuators. The observation made by these sensors are also passed onto the context system and then to the event correlation component on the server side. When an issue is detected, the observation received on the server side is analyzed in order to determine possible control objectives. These control objectives are used by the patch generation component in order to automatically compute a new model of the supervisor (patch) to be embedded in the target application. Once computed, this patch is sent to the client side and is applied using sensors/actuators.
2.2 Requirements for the Self-Healing Component

This section details the features that the self-healing component must implement. It extends on the approach described in [7] by making explicit these feature requirements and their relevance. Although patch generation is part of the FastFix self-healing approach, the corresponding component features are detailed in Section 2.3.

First of all, the FastFix self-healing approach is model-based and models are therefore required in order to perform self-healing. For a system to achieve self-healing principles, it must be able to automatically adapt. Models for possible adaptations are used at runtime for this effect. However, it may be difficult to automatically obtain such models. FastFix aims to provide a fully automated approach where models of the system behaviors are automatically computed from the application implementation. These models are the basis to other aspects of self-healing in FastFix, i.e. both the patch generation and runtime supervision. The models obtained by extraction must be:

1. Complete, i.e. if a piece of data is sensed at runtime, every occurrence of this sensed data should be part of the model.

2. Accurate, i.e. it must contain information that makes it possible to distinguish between different executions.

3. Scalable, i.e. the model extraction must be able to handle large pieces of software.

The completeness of the model is a requirement related to the runtime supervision mechanism. This mechanism aims to keep track of the past execution and decide on whether future possible executions must be prevented. It is then important that the underlying models encodes the relationship between observed sequences of observations and future possible executions. If the model is incomplete and if an unexpected event occurs (an event whose occurrence was not modeled), then it is no more possible to apply supervision as the model not keeps tracks of the past execution of the running program anymore. Therefore, it is important that the model is complete so that no such unexpected event can occur.

In the context of FastFix and the self-healing and patch generation component, model accuracy corresponds to the ability of distinguishing between different executions of the application. For instance, the Finite State Machine (FSM) represented in Figure 2.2 is complete as it can recognize any sequence of events. However, this model is not accurate as it does not make it possible to distinguish between 2 different sequences. Accuracy is an important requirement regarding the patch generation component. As described in [7], automatic patch generation in FastFix is achieved though supervisor synthesis, i.e. an FSM representing a supervisor (or patch) is automatically computed from FSMs modeling the applications behaviors and a property to be ensured on the system. The more accurate is the model of the system behaviors, the more accurate is the generated patch (i.e. supervisor). Although inaccurate patches still ensure the property provided for synthesis, they may prevent too many possible behaviors of the system in order to achieve this. In other words, the more inaccurate the model of the application behaviors, the more functionalities of the application might be prevented unnecessarily at runtime, when applying the patch.
Finally, scalability must also be taken into account when extracting models. Usually, the more accurate is the model, the largest (in terms of state space) it is. One approach that is often considered to handle scalability issues when modeling systems with FSM is to model concurrency between the underlying system components. This is particularly relevant as large systems usually result from the composition of several components.

### 2.3 Automatic Patch Generation Component

As explained in [7] and recalled in Section 2.1, patch generation is achieved in FastFix through supervisor synthesis, relying on the Supervisory Control Theory (SCT). This theory provides conditions and results for the automatic synthesis of models which ensure given properties. In FastFix this is applied in order to automatically generate models of supervisors which will be applied to the applications under maintenance at runtime. These supervisors ensure that executions leading to detected issues cannot occur anymore. Such supervisors are computed from models of the applications and a control objective, which models a property to be ensured (e.g. the avoidance of executions that lead to some issue). Figure 2.3 gives an example of Finite State Machine representing a possible control objective over a set of observable events $A$. This control objective indicates that any sequence of event is acceptable provided that either it only contains one occurrence of event 'b' which occurs at the end, or it contains at least three occurrences of event 'a' so that no 'b' can occur before the first and after the last 'a'.

The main requirements the FastFix patch generation component aims to fulfill are:

1. Supervisor maximality. A supervisor is maximal if it ensures the control objective CO by only preventing executions that would break it, i.e. no un-necessary executions is prevented at runtime.

2. The expressiveness of the Control Objective (CO). Limiting the CO expressiveness may degrade the maximality of the computed supervisor, i.e. it may un-necessarily prevent some possible system executions.
3. Scalability. The models used for SCT are usually state machines. Their size relates to their number of states. The basic algorithms for supervisor synthesis are linear in the size of both the system model and the control objective. Although the synthesis algorithms are quite efficient, they require, in their basic forms, a model of the system represented by a single state machine. As large systems are usually composed of several components, this induces a state explosion problem when computing one single machines by composing the components’ ones. Distributed synthesis takes into account the concurrent nature of the system in order to efficiently compute a supervisor, avoiding the computation of a single machine. Several approaches have been considered in order to achieve this. They all rely on reducing the expressiveness of the control objective or by adding structural constraints on the system (e.g. regarding how the components interact).

2.4 Existing Tools and Libraries

The self-healing and patch generation approaches in FastFix rely on the extraction of Finite State Machines from the application code, and the supervisory control theory that applies on these models in order to generate a supervisor that can be applied as a patch at runtime. Some libraries and tools exist in order to achieve either the model extraction or the supervisor synthesis. It is important to note that to our knowledge, no tool has ever combined both. This gap is one of the main factors that has prevented the application of the supervisory control theory to software systems so far. Being able to automatically extract Finite State Machines from software code will enable the application of this theory and allow for automatic computation of new behaviors.

The Supervisory Control Theory (SCT) was initiated by Ramadge and Wonham in the early 80’s, see e.g. [10]. This theory aims to extends the principle of control theory to discrete event systems. It has mainly been considered at an academic level so far. Therefore libraries allowing for FSM manipulation and supervisory control have been developed by academic groups. Some of these libraries are now described.

- TCT, STSLib (see e.g. [14]) are libraries implemented by the Systems Control Group of the University of Toronto. TCT was the first library implementing algorithms of the SCT. It offers all the features for supervisor synthesis in case of a system modeled as a single FSM. As this approach does not scale well, the STSLib was developed, which library relies on an efficient encoding of the models and provides dedicated algorithms, allowing for supervisor synthesis on larger systems.

- UMDES Lib (see e.g. [13]) is developed by Discrete Event System Group of the University of Ann Arbor, Michigan. It is developed in C and provides similar features and scalability performance as TCT. Moreover, it has an emphasis on decentralized control techniques, i.e. when only partial views of the system are available.

- Supremica (see e.g. [2]) is mainly developed by Department of Signals and Systems of the Chalmers University of Goetborg. It relies on similar principles to STSLib and also takes advantage of the concurrent structure of the system to be controlled. It also allows for simulation of the synthesized supervisor.
SynTool (see e.g. [5]) is developed by the University of Rennes 1 in France as well as by Lero. It offers basic functionalities for supervisory control such as the ones provided by TCT and UMDES but it emphasizes control on concurrent systems, implementing the algorithms introduced in [5] in order to handle large systems.

Regarding tools for model extraction, it is worthwhile noting that manually building Finite State Machines that represent the system behavior is tedious and error-prone. Therefore, approaches for automatically extracting FSM are crucial. For instance, Bandera (see e.g. [4]) allows for model extraction from Java programs. These Finite State Machines consider program variables. The different states of the FSM correspond to different values of the variables of the system. These FSM are therefore state oriented and their number of states depends on the possible value range of the system variables. Moreover, models are only from a subset of Java, leading to incompleteness.

More recently, the authors of [8] considered an approach to extract FSM from several programming languages such as Java and C. Their approach is behavior oriented and considers method calls rather than system variables. The resulting extraction process is lightweight and the size of the extracted FSM remains reasonable for analysis. In this case again, the approach does not consider all of Java constructs and the extracted models are incomplete.

### 2.5 Summary

This section recalled the conceptual models for self-healing and patch generation in Fast-Fix, and the corresponding features that must be implemented: model extraction, patch generation and runtime supervision. It also introduces requirements for these components: completeness, accuracy and scalability for model extraction and runtime supervision as well as expressiveness, maximality and scalability for patch generation.

Section 3 describes the current implementation of the prototype of the self-healing and patch generation components. This implementation is performed taking into account the requirements listed above. The completeness of the extracted models is tackled by following the Java Language Specification ([9]) together with applying over-approximations on possible method calls. Their accuracy is tackled by limiting the use of these over-approximations as much as possible. The maximality of the patch generation is ensured by the supervisory control techniques that are considered and the maximality of the solutions obtained is currently limited the runtime supervision mechanism. Finally these requirements must be balanced with scalability. Section 4 provides details on the current implementation and gives indications regarding the scalability of the features implemented so far.
3 FastFixSH: 1st Prototype Iteration

This section describes the choices made regarding the technology used to develop the FastFix self-healing and patch generation components. It also details the features implemented so far and relate these features to the objectives and challenges presented in Section 2.1.

3.1 Technology and Libraries Used

The several prototypes of the FastFix platform will be developed using the Java language and the OSGi technology. The self-healing component follows this requirement and provides several features, implemented in Java. First, the 1st prototype of the model extraction component is developed as an Eclipse plugin and is detailed in Section 3.2. The first prototype of the patch generation component is based upon the SynTool library and is detailed in Section 3.3. The supervision deployment and runtime supervision mechanisms are respectively detailed in Sections 3.2.2 and 3.2.3. Figure 3.1 represents a screenshot of the eclipse plugin implementing the 1st FastFix self-healing prototype. As shown in the figure, this plugin makes it possible to extract a model of the system as well as to deploy the supervision mechanism into the target application and to generate patches.

Figure 3.1: First prototype of the FastFix self-healing component.
Tools and Libraries Reuse

The models extracted from the Java source code are represented as Finite State Machines (FSM). In order to create and manipulate such models, the self-healing component will rely on the SynTool library (see [5]). As patch generation will also rely on the SynTool library, this will ease the compatibility between the extracted models and the ones required for patch generation. This library, which is developed by Lero, provides the following features.

- Finite State Machines manipulation: this library provides functionalities to manipulate classical FSMs with sets of states (general, initial and final), an alphabet (i.e. set of events) and transitions (i.e. triples of the form \(<state, event, state>\)). SynTool provides usual FSM operations such as determinisation, reachability, co-reachability, product (parallel and synchronous), etc. Variables on the states or transitions are not taken into account by this library.

- Concurrency: this library makes it possible to model concurrent FSMs on top of the usual monolithic representation, i.e. the product of 2 FSMs \(G_1\) and \(G_2\) is not explicitly computed but rather seen as a tuple \((G_1, G_2)\). This view of concurrent FSMs together with dedicated algorithms make it possible to take advantage of the system structure (i.e. concurrency) in order to perform usual tasks on FSM more efficiently. This avoids the construction of a monolithic representation of the concurrent FSMs, which is obtained by performing the product between them. The complexity of computing such a product of FSMs is exponential in the number of states of the FSMs involved in this product.

- Supervisory Control: this library offers classical supervisory control algorithms when the model of the system to be controlled is represented as one single FSM and it also provides algorithms for efficient supervisory control on concurrent FSM. It makes it possible to handle system with several million states ([5]).

3.2 Self-Healing Component

3.2.1 Model Extraction

As the FastFix self-healing approach is model-based and both patch generation and runtime supervision rely on these models, efforts on prototyping the FastFix self-healing component have been mainly put towards the development of the model extraction component. This component has been implemented as an Eclipse plugin (illustrated in Figure 3.1). Figures 3.2a and 3.2b illustrate its principle. Basically, the 1st prototype of the model extraction component considers method invocations and their corresponding method declarations. For instance, if we assume that method1 has been invoked in a piece of code of the target application, then a model of the behaviors of method1 is extracted from its declaration. Figure 3.2a describes the declaration of method1, which invokes method2, method3, method4 and method5. Figure 3.2b represents the FSM extracted from the declaration of method1. It shows that control flow statements such as IF-statements and FOR-statements are taken into account in the extracted model. In order to keep this example simple, it is assumed here that the declarations of method2 to
method5 do not invoke any other method declared in the target application. If it was the case, these method invocations would appear in the FSM of Figure 3.2b as the extraction algorithm works recursively.

Unlike for a classical call graph, the constructed models encodes branching, sequencing and loops. This allows for instance to deduct that method2 always follows method1 and that method5 occurs only after method3 has. Therefore, the extracted models are more accurate than a simple call graph would be.

Figure 3.2: An example of method declaration and the corresponding FSM.

At this stage of the implementation, the value of neither the program variables nor method parameters is taken into account. It is however planed that this will be the case in the 2nd prototype of this component. Therefore method invocations are seen as the basis of our application behavior modeling. First these invocations represent valuable information when debugging an application, as explained in [11]. Moreover, avoiding the variable analysis ensures better scalability of the approach.

The development of the model extraction prototype is done following the Java Specification Language ([9]) as closely as possible. As there is no equivalence between the Java language and Finite State Machines, some approximations of the program behaviors are considered. Of course as completeness of the extracted models is a requirement (as explained in Section 2.2), these approximations must correspond to over-approximations.

The example provided in Figure 3.2 is relatively simple and the 1st prototype of the model extraction component can take many more Java constructs into account:

- Control Flow:
  - IF-statement, conditional expression (i.e. \(a ? b : c\)), Switch-statement, operators \(|, \&, \|, \&\&\).
  - FOR-statement, While-statement, Enhanced FOR-statement (i.e. “for (Object obj: Collection){...}”), enhanced While-statement.
D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

- Break and Continue statements.
- Recursiveness.

- Method invocation occurrences:
  - In any Java expression, e.g. method parameters.
  - Class instance creations, super invocation.

Moreover, in order for the model to be complete, the extracted models also take into account as closely as possible the Java runtime execution mechanisms in specific cases:

- Exceptions.
- Method overriding.
- Class instance creation mechanism, e.g. instantiation of parent objects as well as class members.

Finally concurrency is also taken into account by the 1st prototype of the model extraction component:

- Runnable objects.
- Thread objects.
- Method invocations triggered by event listeners. This allows to capture in the extracted model user interactions such as button clicks.

Taking into account the system concurrency is crucial in order to 1) obtain a complete model and 2) ensure scalability. Representing the composition of 2 concurrent FSMs $G_1$ and $G_2$ as one single FSM $G$ results in an FSM whose number of states is the product of the number of states of both composed FSM, i.e. if $G_1$ and $G_2$ both possess 10 states then $G$ will possess $10 \times 10$ states. This state explosion issue actually constitutes the main bottleneck for applying supervisory control on large systems (together with the absence of FSMs modeling the system behaviors).

Figure 3.3: The model extraction component structure.
Finally Figure 3.3 describes the structure of the model extraction component. First the SynTool library was extended with a FSM factory. This factory aims to create and combine FSMs. It contains methods to:

- create new FSM objects with only one state and no transition (the only state must be initial and can be final),
- create new FSM objects with only two states and one transition (the transition is in between the two states and the source of the transition is the initial state of the FSM. Both states can be final),
- concatenate FSMs, i.e. two FSMs are merged into one so that the final states of one FSM are merged with the initial state of the other one. The resulting FSM represents the concatenation of the languages generated by the concatenated FSMs. This construct is useful to model program statement sequencing.
- variations of the concatenation where the resulting FSM represents an over-approximation of the languages generated by the concatenated FSMs.
- branch FSMS, i.e. two FSMs are merged into one so that the initial states of both FSMs are merged together. The resulting FSM represents the union of the languages generated by the branched FSMs. This construct is useful to model program branching such as IF-statements.
- loop Fsm, i.e. an FSM loops over itself. The resulting FSM represents repetitions of sequences of the initial FSM, i.e. if the initial FSM generates the language $L$ then the resulting FSM generates the language $L^*$ or $L^+$ depending on how the final states of the resulting FSM are set.
- add context to FSM states. This is for instance useful to keep track of states at which Break statements, Continue statements, Exception raises and recursiveness occur.

The Scope and Binding component shown in Figure 3.3 aims to identify the method declarations in the code of the target application as well as the bindings that are necessary to make the link between method invocations and method declarations. Thankfully this part of the component can heavily rely on the Eclipse API for this task. Determining method bindings is a compilation activity, which is challenging for a language such as Java which allows for method overloading and overriding as well as generic parameters and return types. The contribution of the Scope and Binding part of the component mainly resides in making explicit a data structure that allows for easy binding between method invocations and method declarations of methods declared in the target applications.

Finally, the FSM extractor is in charge of parsing the Eclipse AST tree, which models the code of the target application, and building FSMs using the FSM Factory and with respect to the different constructs encountered in the program code.

### 3.2.2 Supervisor Deployment

The previous section described how the prototype for FastFix self-healing makes it possible to extract a model of the behavior of the target application to be FastFixed. This model
D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

will be used in order to compute patches (i.e. new models) for the application behaviors. These patches will be applied by a supervisor on the application at runtime.

At first, the model extracted as described in Section 3.2.1 can be used as a model of a supervisor. In this case, it does not control the target application at all, as it encodes all its possible behaviors. Whenever an issue is detected and automatic patch generation is computed, it results in a new model of a supervisor which will prevent some of the existing behaviors of the application from being executed. In any case, this supervisor/patch must be applied onto the target application at runtime.

As explained in [7], within the FastFix framework, this supervision/patching mechanism will be performed at runtime through sensors and actuators. Therefore patch deployment can here be seen as deploying sensors. These sensors are deployed within the target application and embed a model of the supervisor. They will make it possible to sense information related to method calls, values of method parameters, threads and user input. Moreover, these sensors will also act on the system: they can refer to the embedded supervisor in order to decide whether they should prevent the execution of the observed method call or not. Therefore, the sensors observing the different method calls:

1. can communicate this information to the Context System (as any sensor would),
2. share some data locally (i.e. without referring to the Context System). The model of the supervisor represents such data,
3. and possess some local processing capabilities (i.e. without referring to the Context System), e.g. the ability to decide on whether the execution of a method body should be prevented or not, using the supervisor.

This type of sensors are useful to synchronize the sensing/acting process with the execution of the target application. They enable decision making without the need to communicate with external components, e.g. Context System or Event Correlation component. Such communication would indeed introduce an important overhead when waiting for an external component on whether the method currently called should be executed. However, it is important to note that this approach does not remove the necessity of communicating the sensed values to the Context System. This enables computationally demanding analyses to be performed by external components, e.g. event correlation, fault replication and patch generation component on the FastFix server side. Such communication should be done in an asynchronous manner, i.e. the sensed value can reach the Context System any time after having been sensed.
The 1st prototype of the self-healing component currently implements Points 2 and 3 above. Point 1) will be achieved in the next version of the prototype. Therefore the current version makes it possible to automatically deploy sensors and actuators but cannot yet transmit the sensed values to the Context System.

Figures 3.4 and 3.5 illustrates the supervision mechanism deployment as currently implemented by the prototype, at the source code level. In this example, we consider the
“main” method shown in Figure 3.4. This method calls the “invokeLater” method of the Swing library, which executes the Runnable object given as parameter on the Event Dispatch Thread. The execution performed by this thread is encoded by the “run” method of the anonymous class shown in this figure. This “run” method simply instantiates an object of the class “MainWindow2NoObs” and calls the “setVisible” method on this object. Figure 3.5 illustrates how the supervision mechanism is embedded in the code. For each method declaration in the target application, the body of the method is included in an IF-Statement. The condition of this statement calls the “accepts” method on an object named “controller” and added to the application implementation. It represents the model of a supervisor/patch previously computed. The “accepts” methods takes the name of the thread on which the method is called as well as the method name (the name generated for the compiler for the anonymous class is used in the case of the “run” method, i.e. “MainWindow2NoObs$9412”). The “accepts” method returns a boolean value which indicates whether the supervisor allows for the execution of the method body. If it does, then the body is executed. If it does not, then the execution of the method body is prevented.

Such an instrumentation of the target application code is performed in an automated way, as well as the declaration and instantiation of the static controller object. Its instantiation actually corresponds to loading a file containing a model of a previously computed supervisor/patch.

3.2.3 Runtime Monitoring and Supervisor (Patch) Application

The previous section explains the automated supervision deployment mechanism. Once the controller object of Figure 3.5 instantiates a model of the supervisor, it needs to update its current state which evolves when a method is called and accepted. The FSM automatically extracted from the code of the application must take into account dynamic concurrency in programs, i.e. the number and type of involved FSM varies over time. However, as being extracted from a static artifact (source code), it does not fully achieve this goal. For instance, this model does not indicate which FSM can run simultaneously and on which threads. It is however typical for a software program to run on several threads where the piece of code running on a given thread varies over time as well as the set of current running threads.

In order to take into account the dynamic related to threads and graphical components, we implemented a specific FSM execution mechanism. Unlike the standard execution mechanism on FSMs, it considers that only a subset of the FSMs may run simultaneously. This mechanism is embedded in the implementation of the supervisor that is deployed as described in Section 3.2.2 and consists in

- mapping at runtime the observed current thread and method call to the appropriate running FSM in order for it to update its current state,

- mapping at runtime the observation of a triggering event, i.e. the first event that can be triggered from a FSM such as “main” methods, “run” methods for threads and “actionPerformed” methods, to the corresponding FSM, even when it is not already running, i.e. it is currently not involved in the program execution.

Moreover, it is important to note that the supervisor embedded in the FastFix target application is a synchronized object and it is therefore safe to call it form different
threads. Such an approach makes it possible for the supervisor to make decision about the concurrency of the target application, e.g. deadlock avoidance. However, this introduces some extra concurrency between threads, i.e. threads have to share an extra resource: the supervisor.

Finally in FastFix, patches that are generated automatically correspond to supervisor models and are applied to the target application by controlling the application behaviors according to these models. In order to achieve this, the FastFix self-healing component relies on the SynTool library (see e.g. Section 2.4) and extends it. However, as explained in Section 3.3, these generated models are different from the ones automatically extracted and that are considered in this section. Therefore some adaptation of the existing supervisor synthesis algorithm were considered in order to overcome this issue.

### 3.3 Automatic Patch Generation Component

Patch Generation is actually part of the self-healing approach. It corresponds to combining the model of the target application with the observations of undesired behaviors in order to create a model of a patch that will prevent future occurrences of these undesired behaviors. The models manipulated by this component are Finite State Machines (FSM) and it relies on the FSM library SynTool (e.g. see Section 2.4) for supervisory control. This library contains algorithms to manipulate FSMS and compute models of supervisors/controllers, which can be used as patches in FastFix. This library is particularly suitable for systems modeled as a composition of concurrent sub-systems. However, it considers that the components running in a concurrent manner is fixed over time. Therefore, it does not deal with dynamically changing thread processing, unlike the runtime model presented in Section 3.2.3.

Regarding the current implementation of SynTool’s supervisor synthesis (i.e. automatic patch computation), it does not handle the case where several instances of the same process features run in parallel and especially when the number of such instances that can run in parallel is unknown during analysis. However, this can for instance occur when threads are created and started in the body of a loop, or whenever a user presses a specific button.

Although this features will be implemented in the course of the project, this first prototype is limited to the implementation of an intermediary case. This corresponds to the case where no more than one instance of each extracted FSM can be executed at a given time and the control objective must describe whether method calls occur on similar threads or not. This already improves the current implementation of SynTool has it allows for not all the FSM to be executed at the same time and for an FSM to be executed on different threads over time (provided that the previous executions were completed). This new feature is achieved by:

- Implementing a renaming of the labels of the FSMs automatically extracted, i.e. making these FSM behaving in an asynchronous fashion with respect to each other.
- Implementing a FSM transformation that generates repeated behaviors of the initial FSM. This is done with a similar algorithm to the one that creates a looping FSM in the FSM factory presented in Section 3.2.1. This transformation is important in order to generate patches that take into account that the execution of some methods may be repeated on different threads over time.
Once these transformations are performed on the FSMs that were automatically extracted, then the SynTool supervisors synthesis functionality can be applied in order to produce a model of a supervisor. This approach ensures the requirement regarding the maximality of computed solution, i.e. the solution is the most permissive supervisor ensuring the control objective on the system model.

Finally, the generated model does not match the one presented in Section 3.2.3 (e.g. regarding its concurrency and dynamic). Therefore a mapping between these two types of model was also implemented so that the supervisor could be consulted in an appropriate way at runtime, i.e. so that the computed patch could be applied at runtime.

### 3.4 Summary and Limitations

Section 3 describes the implementation of the model extraction, runtime supervision and patch generation mechanisms. This implementation is driven by the requirements provided in Section 2.

- In order to achieve completeness, the implementation of the model extraction component considers the Java Specification Language in order to identify and take into account all the Java constructs from which methods can be called.

- Although over-approximations are necessary when extracting models of method calls, this implementation takes accuracy into account by avoiding unnecessary approximations.

- The program concurrency is taken into account in order to limit the size of the extracted model. Concurrent Finite State Machine are extracted instead of a single one.

- Specific supervisor/patch synthesis for concurrent FSM are applied in order to tackle the scalability issue that may raise during patch generation.

- Moreover, this synthesis algorithm ensures not only the patch correctness but also its maximality (i.e. it restricts the behaviors of the application model as little as possible).

However, the current implementation of this first prototype induces some limitations:

- The expressiveness of the control objectives is limited to properties that do not involve two different instances of the same method running on different threads.

- The libraries and other project dependencies of the target application to be Fast-Fixed must not invoke methods from this application, i.e. only one-way dependencies are authorized (the components that the target application depends on must not depend on the target application).

---

1The implementation of this mapping is part of component in charge of encoding the runtime models, more than the patch generation component. However, this mapping is strongly dependent on the type of models that are generated.
• Completeness of the extracted model is not guaranteed if some graphical components do not run on the event-dispatcher thread. Best practices in Java actually require so, through the use of the SwingUtilities.invokeLater method.

• The initial method from which the FSMs are built (usually the "main" method) should not be in the default package but in a named one.

• It cannot include in the FSM extracted models, events corresponding to the invocation of super constructors. It is important to note that the FSM modeling the execution these super methods are still extracted. Only the event representing the call of these super methods are not part of the model. This limitation is expected to be relieved in a future version of the prototype.
4 Results and Next Steps

4.1 Illustrative Example

This section illustrates the use of the 1st prototype of the FastFix self-healing component. It considers a rather simple example and focuses on the main concepts of the approach: model extraction, supervision deployment and patch generation. The example under consideration is the one of a calculator implemented in Java and using the Swing Library. This calculator implements basic operations on integer. However, the exception that may be raised from dividing by zero is not handled. It is assumed in this example that this issue is present in the deployed application, i.e. it was not caught during the testing phase and its presence is therefore unknown. Although the error presented in this example is quite simple, it is characteristic of a larger class of errors, e.g. where some exceptions are not handled and are raised whenever the user performs actions in an order that the developer has not planned for.

![Figure 4.1: An illustrative Calculator Example.](image)

In this example, we illustrate how the 1st prototype of the FastFix self-healing component can be used in order to overcome the possible occurrences of issues at runtime. Figure 4.2 shows a screenshot of the Eclipse plugin that was developed and that implements this first prototype. This Plugin provides a “Self-Healing” menu which gives access to the three main functionalities of the FastFix self-healing approach (see Section 2.1): model extraction, supervision mechanism and patch generation.

First, models must be extracted from the code of the FastFix target application, i.e. the calculator in this case. By clicking the “Extract FSM” item of the self-healing menu in Figure 4.2, the FastFix user automatically retrieves Finite State Machines (FSM) representing the behaviors of the application, in terms of method calls. This extraction takes into account Swing components (more specifically the concurrency introduced by the event listeners) and thread creation (in this case, and following good practices in Java Swing, the graphical components are instantiated on a dedicated thread). Moreover, this
D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

extraction is performed from the “main” method of the target application, as shown by the selection of this method in the Eclipse Outline View, on the right hand side of Figure 4.2. The FSM computed in this case encodes all the possible sequences of method calls that can occur at runtime from calling this main method. 18 FSM were extracted in this case: 1 for each of the event listeners associated to the 16 buttons of the application, 1 for the main method and 1 for the execution of the thread instantiating the graphical components. These FSMs represent the possible sequence of method calls that can occur from calling the main method, or clicking one of the buttons, or starting the Event Dispatch Thread on which are instantiated the graphical components.

Figure 4.2: Eclipse implementation of the prototype of model extraction approach.

Figure 4.3 represents the FSM extracted from the “actionPerformed” method of the event listener associated to the “divide” button. As the application is relatively simple, it is not surprising that the FSM obtained also are quite simple. This FSM indicates that whenever the actionPerformed method is called, it is followed by the call of a method called “process”. The “process” method is actually in charge of processing the values entered into the calculator according to the value of the operation button which is clicked.

Section 4.2 provides preliminary results regarding the scalability of the 1st prototype of the FastFix model extraction component.
D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

Once the Finite State Machine representing the behaviors of the target application are extracted, then the supervision mechanism must be deployed in the application (see Section 3.2.2). This action is performed automatically by clicking the “Deploy Supervision Mechanism” item of Figure 4.2. For this 1st prototype, deployment of the supervision mechanism is achieved through source code instrumentation as illustrated in Figure 4.4. This instrumentation consists first of automatically introducing a controller (i.e. supervisor) object in the program. Before any patch is computed, this supervisor is first instantiated as to implement the model of the application which was previously extracted. Then instructions are added to methods in order to surround their bodies with IF-THEN statements. The condition evaluated in the IF part corresponds to the acceptance by the supervisor of the execution of the method body (in Figure 4.4, this corresponds to calling the method “accepts” on the controller object). If the controller accepts the method execution, then its body is executed. If it does not accept the method execution, then no statements are executed at all. The call of the accept method not only returns the decision made by the controller at a given point of the program execution, it also updates the current state of the model to the one reached by accepting the current method execution. This makes it possible for the controller to take decisions according to the past execution.

Figure 4.3: Finite State Machine automatically extracted and representing the possible executions from pressing the “Divide” button.

---

\(^2\) Other approaches could be considered here, such as displaying a warning message to the user.
Once models have been extracted and the supervision mechanism has been deployed, then the application can itself be deployed to users environment. Thanks to the embedded supervisor, the deployed application can monitor itself. When a user attempts a division by zero, such an exception is raised, exhibiting the need of a patch. A patch is obtained by automatically synthesizing a model of a new supervisor, using the previously extracted model of the application as well as a control objective. In future versions of the FastFix prototypes and whenever it will be possible, this control objective will be computed from the sequence of method calls that lead to the observed exception and will be provided by the correlation system. As for now, it is designed manually and is represented in Figure 4.5. Because of the size of its labels, this Finite State Machine is only partially represented in this figure. However, it is a relatively simple machine as it only contains three states (2 of them are represented in Figure 4.5). This FSM simply describes that the actionPerformed methods associated to operator buttons, i.e. $+, -, *, /$, should not occur after the user has clicked the “divide” button followed by the “zero” button.
From the models previously extracted and the control objective of Figure 4.5, a new model of a supervisor is automatically computed. This is achieved by clicking the “patch Generation” item of the menu presented in Figure 4.2. This computation relies on an extension of supervisory control algorithms, as explained in Section 3.3. These algorithms take into account than not any method body execution can be prevented (i.e. not every method is controllable). In order to ensure the system stability at runtime, only methods that are not called from other methods are assumed to be controllable, i.e. the main method and all the actionPerformed methods associated to button clicks in this example. Preventing the execution of a method bodies that are called form other methods may indeed lead to issues. The prevented execution could compute new objects and assign others with values that are expected later in the execution. The concept of controllability of the supervisory control theory is therefore extremely useful in order to guarantee the correctness of the computed patch.

Once this new supervisor, i.e. patch, is computed, then the file containing the previous model of the supervisor on the user environment is replaced with the newly computed one. When the target application is relaunched, this new file becomes the one from which is instantiated the controller object that appears in the code of Figure 4.4. With this controller, the execution of the body of the actionPerformed method is prevented whenever the user has just clicked the “divide” button followed by the “zero” button. It is important to note that the execution of the body of this actionPerformed method is executed normally in any other cases, hence only preventing executions that can lead to an exception.

4.2 Scalability Results

In this section, we present preliminary results on the model extraction component. More specifically, this component was applied to three different target applications from the open source community: Avrora, Sunflow and JEdit. Avrora is a processor simulator and is part of the Dacapo benchmark ([3]). Sunflow is a rendering system for image synthesis and is also part of the Dacapo benchmark ([12]). Finally JEdit is a programer’s text editor ([1]). It is an Eclipse-like editor which can be extended with plugins. Avrora, Sunflow and JEdit are all written with the Java language. Figure 4.6 illustrates the application of the model extraction component to the Sunflow system. In this figure an "actionPerformed" method associated to a task cancellation button is selected. Figure 4.7 shows the generated FSM for this actionPerformed method. First the method “taskCancel” is called, which itself calls a “printInfo” method which itself calls a “print” method.
This section provides scalability results for the current implementations of the model extraction and the patch generation components. Runtime supervision scalability study requires that the extracted models are complete. Although our methodology follows the Java Specification Language in order to ensure this, it is not fully achieved yet and improvements must be made in this direction. Therefore the results provided in this section focus on the model extraction and patch generation. These evaluations were performed with a MacBook Pro Dual Core i7, with a frequency of 2.66 GHz and 4GB of RAM.

Table 4.1 summarizes results obtained when applying the model extraction component to the three different target applications: Avrora, Sunflow and JEEdit. It contains information regarding the target applications as well as times taken by the different model extraction computation phases.
First of all, the amount of method declarations is provided for each target application, as an indicator of their size. Avrora and JEdit possess a rather similar amount of method declarations (around 8000), while Sunflow only possess 1500 ones. The model extraction consists of several phases, all reported in Table 4.1: Scope and Binding computation, FSM construction computation, FSM determinization computation and FSM saving computation. From a user’s point of view, the total time taken to obtain a file containing the model of the system is represented by the “Total Time”, which sums up the times obtained for each computation phases.

The Scope and Binding computation represents the analysis of the code that is first performed in order to limit the model extraction to method invocations whose corresponding method is declared by the target application, e.g. methods declared in java.* libraries are not taken into account in the model extraction. The FSM computation corresponds to going through all the method invocations from a given point in the program, taking into account the program control flow, and building the corresponding FSMs. This process also takes into account concurrency, with the computation of separate FSM for the different threads, runnable objects and event listeners instantiated by the application.

FSM determinization is known for not being efficient in the worst case: it is exponential in the size of the FSM to be determinized. However, this complexity depends on the amount of indeterminism of the initial FSM, which in our case is partly introduced by the FSM construction process. The results obtain in Table 4.1 suggest that the efficiency of determinizing the constructed FSMs is of similar order as building these FSMs.

Finally the necessary time for saving the determinized FSMs is provided as hard drive access may induce a quite important overhead in the process, as shown in the case of JEdit where there are numerous FSMs to be saved and where three of them possess more than 1000 states (the largest one possessing about 2600 states).

Overall, the approach for model extraction appears to be feasible for applications with several thousands method declarations and generating up-to at least 80 FSMs. This corresponds to gigantic state spaces for these models: Sunflow has at least $2^{27} \times 527$ states while JEdit has at least $2^{83} \times 2588$ states! As the model extraction is performed off-line and is a once-off computation, a duration of several minutes for it is acceptable.

<table>
<thead>
<tr>
<th></th>
<th>Avrora</th>
<th>Sunflow</th>
<th>JEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Declarations</td>
<td>8889</td>
<td>1508</td>
<td>7420</td>
</tr>
<tr>
<td>Scope and Binding Time</td>
<td>16236 ms</td>
<td>3555 ms</td>
<td>22056 ms</td>
</tr>
<tr>
<td>FSM Computation Time</td>
<td>1269 ms</td>
<td>981 ms</td>
<td>112279 ms</td>
</tr>
<tr>
<td>FSM Determinisation Time</td>
<td>355 ms</td>
<td>384 ms</td>
<td>126256 ms</td>
</tr>
<tr>
<td>FSM Saving Time</td>
<td>84 ms</td>
<td>991 ms</td>
<td>202604 ms</td>
</tr>
<tr>
<td>Total Time</td>
<td>17.94 sec</td>
<td>5.91 sec</td>
<td>7 min 43 sec</td>
</tr>
<tr>
<td>Number of FSMs</td>
<td>1</td>
<td>28</td>
<td>84</td>
</tr>
<tr>
<td>Largest FSM</td>
<td>201</td>
<td>527</td>
<td>2588</td>
</tr>
<tr>
<td>Smallest FSM</td>
<td>201</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Model (Patch) Size</td>
<td>37 KB</td>
<td>406 KB</td>
<td>9.6 MB</td>
</tr>
</tbody>
</table>

Table 4.1: Preliminary results on FSM extraction scalability.
The size of the generated models is also considered. Table 4.1 indicates for each target application, the amount of FSMs generated, the size of the largest and smallest FSM generated and finally the size of the file containing these FSMs. This table shows that although Avrora possesses many more method declarations than Sunflow, the model generated for it, is substantially smaller (201 against an estimated number of states of $2^{27} \times 527$). This shows that although the amount of method declarations in a target application is an interesting indicator, it is not sufficient in general to estimate the complexity of the generated models. Also, the size of the files containing the generated models can itself vary substantially. As shown in Table 4.1 it ranges between 37KB to nearly 10MB, for the three target applications considered.

<table>
<thead>
<tr>
<th>Control Objective Number of States</th>
<th>Avrora</th>
<th>Sunflow</th>
<th>JEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.172 s</td>
<td>2.015 s</td>
<td>5 min 41 s</td>
</tr>
<tr>
<td>10</td>
<td>0.363 s</td>
<td>3.397 s</td>
<td>9 min 34 s</td>
</tr>
<tr>
<td>50</td>
<td>0.895 s</td>
<td>11.533 s</td>
<td>42 min 57 s</td>
</tr>
</tbody>
</table>

Table 4.2: Results on patch generation scalability.

Finally, Table 4.2 provides scalability results regarding the patch generation component. The purpose of this evaluation is to obtain indications about the scalability of computing patch, with respect to the control objectives size. In order to achieve this, random FSMs with a given number of states were generated. Although randomly created, these control objectives follow a similar structure to the one of the control objective of the calculator example of Section 4.1, i.e. they accept any trace observed from the target application except for a the one finishing with a specific sequence. In the example of Section 4.1, every sequence except for the ones finishing by the sequence “divide”, “zero”, “equal” (or “divide”, “zero”, “plus”, etc) are accepted. In this example, the corresponding FSM possesses three states. For this evaluation, we consider control objectives encoding a similar pattern where the length of the finishing sequence has 5, 10 or 50 states. Although more evaluation will need to be performed to this respect, it is expected that control objective will be encoded with FSM with less than 50 states, as the number of states in this case represents the number of relevant events necessary to characterize the application traces that are not desired, i.e. events involved in describing a pattern of a bad trace.

The algorithm used for patch computation are implemented in SynTool and described in [6]. It takes advantage of the concurrent nature of the model and computes one supervisor over each component instead of a global one. The complexity of this algorithm is linear in both the size of the component and the control objective. Table 4.2 indicates the time required to compute patches for the generated control objectives.

The case of Avrora shows the time taken for a model with one single FSM of about 200 states. It takes less than a second to compute a patch, even for a control objective that possesses 50 states. As shown with Sunflow application, it takes only a few seconds to compute a patch for a model consisting of 28 FSMs where some of them have a few hundred states. The model of JEdit is at a different scale with more than 80 FSMs with several of them having more than 1000 states, the largest one possessing about 2500 states. The computation of a patch for this application takes several minutes for control objectives with less than 10 states and around 43 minutes for a control objective.
D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

with 50 states (the computation of a patch for the FSM with 2500 states takes about 16 minutes). These figures demonstrate the feasibility of applying supervisory control on models extracted from software code. It is important to note that these figures can be improved by combining the algorithm used in Table 4.2 to other approaches which allow for instance to select a subset of FSM on which to compute patches.

4.3 Next Steps

The work that will be conducted after this implementation of the 1st prototype of the FastFix self-healing component follow three different directions.

First, an immediate step is about more evaluation and improving the current prototype. Especially, some evaluation of the overhead induced by the supervision mechanism must be performed. In order to achieve this, we consider the Dacapo benchmark which provides open source applications and associated test cases. Although these test cases do not exhibit any issue of these applications, they make it possible to reproduce their executions. A comparison of the time required to reproduce these executions with and without deployment of the supervision mechanism will provide an estimation of the overhead induced by our approach.

Secondly, an important next step is about interaction and better integration of the self-healing component with the other FastFix components. At present, all these components fit within a unique OSGI architecture. However, no interaction between the self-healing component and others is available yet. One of the next steps towards such interactions concerns the unification of the sensing systems, i.e. the context system sensors and the embedded supervisor. Another aspect is the communication between the event correlation and the patch generation components so that control objectives can be automatically provided to the patch generator. These considerations about component interactions must also fit in the client-server architecture of the FastFix platform.

Finally, a third direction to consider is the improvements of the self-healing component itself, through research. This research must include the management of dynamic concurrent systems in order to overcome the issue raised in Section 3.3. It should also include the management of the values of some of the system variables and in particular of the user input, in order to drive the patch generation, e.g. the execution of some methods may be prevented depending on the values of their parameters.

The second iteration of the FastFix self-healing prototype will aim to tackle these different aspects.
Bibliography


D6.3: 1st Prototype of the Self-Healing and Patch Generation Component

