ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

Project number: 258109
Project Acronym: FastFix
Project Title: Monitoring Control for Remote Software Maintenance
Call identifier: FP7-ICT-2009-5
Instrument: STREP
Thematic Priority: Internet of Services, Software, Virtualisation
Start date of the project: June 1, 2010
Duration: 33 months

D6.5: 3rd Prototype of the Self-Healing and Patch Generation Component: FastFixSH

Lead contractor: Lero
Author(s): Benoit Gaudin, Zhanming Cui, Pierre Monjallon, Mike Hinchey.
Submission date: August 2012
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 the FastFix D6.5 deliverable, i.e. implementation of the 3rd prototype of the FastFix self-healing and patch generation components. It aims to present improvements brought to the FastFix self-healing component, as planned in the course of the project and since the version described in deliverable D6.3 [5]. For this iteration of the prototype, improvements relate to the addition of variables into different aspects of the component: model extraction, supervision mechanism, runtime supervision as well as patch generation. This makes it possible to consider the target application variables and user inputs during self-healing. This improves on the previous iteration as models are finer grained and therefore lead to the generation of more accurate and relevant patches. This document first lists improved features in Section 1. Then Section ?? describes a typical use of the prototype. Finally [2] presents some results regarding time and memory scalability and performance.
1 Development Status

This section lists the improvements and additions performed on the FastFixSH self-healing component developed as part of WP6. The self-healing approach undertaken in FastFix consists of different phases: model extraction, supervision deployment, runtime supervision and patch generation. The previous prototypes of FastFixSH tackled aspects of each of these phases.

1.1 Recall of FastFix Self-Healing Approach

Figure 1.1 recalls the overall FastFix self-Healing approach. The implementation of the different phases involved in this approach are detailed in this section.

![Figure 1.1: The FastFix Self-Healing Approach.](image)

1.1.1 Model Extraction

Model Extraction corresponds to Phase 1 in Figure 1.1 and is a step that aims to automatically obtain models of the source code from which analysis on the application behaviors can be performed, such as patch generation.

In the 2nd prototype of FastFixSH, the model extraction feature builds Finite State Machines (FSM) from Java source code. This mechanism takes into account most Java constructs as described in [5].

The model extraction mechanism automatically selects triggering events, also called entry points. Intuitively triggering events aim to encode methods declared in the target application and whose execution is triggered from the environment (user, execution environment, etc). Each triggering event also corresponds to one single FSM of the extracted
models. This FSM represents the behavior that can be observed from the invocation of the triggering event. The output of the model extraction mechanism provides one FSM for each of the triggering events. These FSM are used at runtime for supervision and must be deterministic in order to achieve this.

### 1.1.2 Supervision Deployment

Supervision Deployment corresponds to Phase 1' in Figure 1.1 and is a step that aims to instrument the code of the application in order to embed supervisors that control the application behaviors at runtime.

The supervision deployment mechanism modifies the body of all the method declarations of the methods involved in at least one of the extracted FSM. As shown in Figure 1.2, the source code is modified so to condition the execution of a method body to the acceptance of a controller. The method body is also modified so that exceptions can be caught by the controller.

```java
// Puts the given status in the error log view.

public static final String STATUS = "LOADING ERROR: 

try {
    // Eclipse platform displays IllegalArgumentException instead of throwing it.
    // We'll handle this by throwing it ourselves.
    if (status == null) {

    } catch (java.lang.Exception e) {
        // Handle exception here.
    } finally {
        org.eclipse.emf.compare.EMFCompareException controller = null;
    }

    // ... other code...
}
```

Figure 1.2: Example of Supervision Deployment on a method from the Moskitt application.

Annotations were introduced to encode results of the different phases of the model extraction and supervision deployment mechanism. These annotations are used to store information about the different phase of the supervision deployment mechanism: method identification, entry points computation and instrumentation. Three different types of annotations are introduced for each of these different phases: `Id`, `ToSupervise` and `IsSupervised`. These annotations are automatically added to the application source code.

As for the model extraction mechanism, automatic annotation relies on the data structures of the Eclipse platform for source code representation. AST trees are programmatically modified so that annotations are added to method declarations.

### 1.1.3 Runtime Supervision

Runtime Supervision corresponds to Phase 2 in Figure 1.1 during which the supervisors embedded in the application control its behaviors and detect the raise of un-handled
exception.
At runtime the instrumented application is executed instead of the initial one. The static controller is first instantiated, loading models computed from the models previously extracted. When a method is invoked during execution, the “accepts” method is called on the controller in order to check whether the execution of the body of the method being executed is allowed by the model. If it is the case, then the method body is executed, otherwise the method returns. In this version of the prototype the supervisor only accepts the execution of a method body if and only if the model of the system allows this method call from the current state.

1.1.4 Patch Generation

Patch Generation corresponds to Phases 2’ and 3 in Figure 1.1 where analyses are performed from the extracted models and the logs generated after an un-handled exception occurs. The outcome of this analysis are new models for the supervisors embedded in the application at runtime.

When an exception is raised, a patch generation mechanism generates a control objective, i.e. a FSM encoding a property (set of behaviors) to be ensured at runtime.

From this control objective and the model extracted from the application source code, a new model of the system behaviors is also generated. This model contains the behaviors of the initial one, except for the ones that lead to the observed exception. Unlike the control objective, this model takes into account the fact that some method executions cannot or should not be prevented.

Expertise on the technology in which the target application is developed is required by the patch generation algorithm in order to characterize methods that are directly invoked from user interactions. In case of the calculator example, these methods were the actionPerformed methods of the EventListener objects. In order to comply with the technology considered for the FastFix scenarios, the current implementation of the patch generation algorithm considers Eclipse technology. For large systems, it is not reasonable to consider a manual selection of these methods. Therefore we consider the name of the methods and their declaring classes in order to specify relevant methods. For Eclipse technology we focus on methods whose name contains run and their declaring class name contains action as well as methods whose name contains execute and their declaring class name contains command.

1.2 3rd Iteration of FastFixSH

Section 1.1 recalls the features implemented in the two first iterations of FastFixSH. In this section, we present the improvements and modifications leading to the third prototype of FastFixSH. As planned in the FastFix Description of Work document, these improvements aim to introduce the concept of variables into the approach. Variables are introduced in the model extraction, supervision deployment, runtime supervision and patch generation.

For scalability and complexity reasons, our approach does not consider raw values of the variables but rather an abstraction of these values. The concept of domain abstraction is presented in Section 1.2.1
1.2.1 Domain Abstraction

This section presents the notion of Domain Abstraction which is considered in FastFixSH in order to abstract variable values. This approach is inspired from works such as Abstract Interpretation used for static analysis [3], as well as from the dynamic analyzer Daikon [4]. The main idea of domain abstraction is to reduce large and possibly infinite concrete domains into small and finite ones.

In order to illustrate this, we can consider the set $\mathbb{Z}$ of integers as a concrete domain. This set is infinite and contains positive and negative numbers. One simple abstraction of $\mathbb{Z}$ which is used in Daikon [4] is

$$A_{\mathbb{Z}} = \{\mathbb{Z}^- , \{0\} , \mathbb{Z}^+\}$$

where $\mathbb{Z}^-$ represents the set of strictly negative integers, $\mathbb{Z}^+$ represents the set of strictly positive integers and 0 represents the neutral element of the integer set. $A_{\mathbb{Z}}$ is a set containing only three elements and is an abstraction of the set $\mathbb{Z}$.

Domain Abstraction are useful when combined with an abstraction function. Such a function aims to map any value of a concrete domain to a value of the abstracted domain. In the case of $\mathbb{Z}$ and abstraction $A_{\mathbb{Z}}$, a natural abstraction function $f$ can be defined such that for $x \in \mathbb{Z}$,

$$f(x) = \begin{cases} 
\mathbb{Z}^+ & \text{if } x > 0 \\
\mathbb{Z}^- & \text{if } x < 0 \\
\{0\} & \text{if } x = 0 
\end{cases}$$

FastFixSH considers several domain abstractions for different data types. These abstractions are implemented in the eu.fastfix.common.selfHealing.supervisoryControl.valueAbstraction package. New abstractions can be easily added to FastFixSH by adding new classes to this package in order to extends AbstractedObject (i.e. domain abstraction) and implement IAbstractor (i.e. mapping between concrete and abstract domains). Currently FastFixSH contains the following abstraction domains:

- AbstractedObject: \{Null, Not_Null\}
- AbstractedDouble: \{Null, Negative, Zero, Positive, NaN, Infinite\}
- AbstractedChar: \{Null , Not_Zero, Zero\}
- AbstractedBoolean: \{Null , True, False\}
- AbstractedInteger: \{Null, Negative, Zero, Positive\}
- AbstractedString: \{Null, Empty, Not_Empty\}
- AbstractedCollection: \{Null, Empty, Not_Empty\}
- AbstractedArray: \{Null, Empty, Not_Empty\}
- AbstractedMap: \{Null, Empty, Not_Empty\}
- AbstractedList: \{Null, Empty, Not_Empty\}
- AbstractedVoid: \{Null\}
Abstractors which makes it possible to map a concrete value onto an abstract one are also implemented in FastFixSH and used in different phases of the FastFix self-healing approach.

1.2.2 Model Extraction

For this 3rd iteration of FastFixSH, variables are now part of the extracted model. More precisely given a method declaration, FastFixSH extracts information about the parameters of this method as well as the attributes of the classes in which this method is declared (not only the declaring class but also classes that contain the declaring class, in case of nested classes). Not every method declaration that is part of the extracted models contain information about their parameters and attributes. Instead this treatment only applies to entry points (i.e. triggering events).

This is due to the nature of entry points. When computed automatically, entry point methods represent methods that lead to any observed method. In other words, an observed method call is always the consequence of the observation of the last entry point method (for methods that occur on the same thread). We rely on the fact that in general, the value of the parameters of an entry points as well as the class attributes characterize the following sequence of non entry methods as well as their parameters. Of course, because of approximations made during model extraction as well as domain abstraction and program non determinism, this is not always the case. However we believe that this assumption is sufficiently relevant to be implemented in FastFixSH. Moreover, this assumption helps control the runtime overhead induced by the supervisor. Supervisor need indeed to check the variable abstract value in order to make decisions on the methods to be executed at runtime.

1.2.3 Supervision Deployment

In this 3rd iteration of FastFixSH, the supervision deployment mechanism also takes into account variables of the target application. Compared to the previous version of FastFixSH, the source code is instrumented so that the supervisor that is embedded in the application not only conditions the execution of a method to the value of its ID (which corresponds more or less to its signature) but also the abstract value of its parameters and class attributes. For this, the accepts method of the DynamicController class (which represents runtime supervisor object) has been modified in order to take into account any number of parameters. The last objects that are passed as parameters to this method represent the method parameters and class attributes together with an Abstractor object which makes it possible for the supervisor to compute the appropriate abstracted value of this parameter at runtime. Figure 1.3 illustrates the use of the accepts method.
D6.4: 2nd Prototype of the Self-Healing and Patch Generation Component

Figure 1.3: An example of supervision deployment taking into account method parameters and class attributes.

1.2.4 Runtime Supervision

In the 3rd iteration of FastFixSH, method parameters and class attributes are evaluated by the supervisor against its own model. For this, the supervisor computes at runtime the abstraction of the current value of a parameter or attribute, according to its type and the abstract domains listed in Section 1.2.1. This abstracted values are then compared to what is authorized in the model of the supervisor. If this model allows for the execution of a method with the corresponding abstract values, then the method is executed. However, if this abstract values are not authorized for this method, then the supervisor prevents the execution of the methods. This improves on the previous version of FastFixSH as embedded supervisors only prevented method execution according to their signature. Conditioning method executions to the value of their parameters and class attributes offer a finer supervisor decision process.
1.2.5 Patch Generation

Variables are also taken into account in the patch generation process. Figure 1.4 represents a simple example of traces recorded by the self-healing component embedded in the target application at runtime. This example illustrates the case of a trace with 3 method calls, named a, b and c, with an integer parameter, named X.

![Figure 1.4: An example of trace obtained from runtime monitoring of method call and variables.](image)

Previous versions of the patch generation mechanism did not take into account method parameters. This feature has been added to the component in this iteration. The patch generation algorithm follows the same principle as the previous one, i.e. it performs a backward search in an observed trace leading to an issue until the tail of this trace does not correspond to a trace observed in acceptable traces (obtained from testing for instance).

In order to illustrate this approach, we consider that the trace of Figure 1.4 leads to an issue, e.g. an exception is raised just after the call of method c with parameter X being negative. A coarse way of avoiding this issue from occurring again is to avoid the execution of method c with a negative value of X. This corresponds to the controller represented by Figure 1.5 and with the following control function: “if the control objective is in state 0 and method call is c and X < 0, then prevent the execution of c”.

![Figure 1.5: A basic control objective preventing the execution of](image)

Now if we assume that a call of method c with X < 0 has been observed in previous traces that did not lead to an issue, then we can consider a more refined control objective such as in Figure 1.6, with the following control function: “if the control objective is in state 1 and method call is c and X < 0, then prevent the execution of c”. Note that in this case, a call to method c is only prevented if X < 0 and a call to method b with X > 0 has been previously observed. We assume here that a call to method b with X > 0 followed by a call to method c with X < 0 has not been previously observed in an acceptable trace. In the case where it has, then the patch generation mechanism can continue and a controller (control objective + control function) involving methods a, b and c would be considered.
Finally the 3rd iteration of the FastFix self-healing component also implements a new patching strategy. In previous iterations of the self-healing component, the patching strategy only applied to traces in leading to issue because of the sequencing of the observed method calls. This case unfortunately does not take into account issues that occur because of the absence of some method calls. In order to tackle this problem, we consider alternating patterns such as presented in Figure 1.7. This patterns indicate that calls to a and b always alternate. In particular, this entails that a call to b should not occur unless a has been called since the last occurrence of b.

This case models behaviors such as the initialization of a variables x (for instance by calling a method a) before a method b is called. When the application enters a behaviors where x is not initialized first, then an error occurs. Moreover, the case where x is initialized and b is not called is also suspicious and considered as an incorrect behaviors in our approach.

Such alternating patterns can be computed from collected traces obtained from test executions and representing correct application behaviors. Once computed, these patterns are embedded in the target application and used for monitoring. If the execution ends and the pattern is not in a final state (i.e. double-circled state), then the pattern is said to be broken.
Figure 1.8: An control objective derived from a broken pattern.

Information about broken patterns is sent to the FastFix server and used by the patch generation component in order to generate the control objective as presented in Figure 1.8. This control objective indicates that method b cannot be called before a is called, hence ensuring proper alternation. It is important to notice that algorithms for the computation of alternating patterns usually return a substantial amount of false positive. For this reason, controlling behaviors representing patterns as in Figure 1.8 may be too restrictive. On the other hand, the occurrence of an error is a strong indicator that a pattern is actually a true positive and should therefore be enforced.

1.3 Summary
# Results

In Deliverable D6.3 results regarding the scalability of the model extraction, supervision deployment mechanisms and patch generation were presented. In Deliverable D6.4 we presented results regarding the runtime overhead of the self-healing approach. In order to evaluate this overhead, the Dacapo benchmark [2] was considered. In this section, we extend results on the runtime overhead of our approach in the case of the Moskitt target application. This overhead is computed both with and without taking into account the program variables. Therefore the presented results make it possible to evaluate the overhead induced by supervision with and without variable monitoring as well as to compare the overhead induced on Moskitt with respect to the applications considered in the Dacapo benchmark.

As for previous deliverables, experiments presented in this section were conducted on a MacBook Pro with a 2.66 GHz i7 dual core processor and 8 gigabytes of RAM.

## 2.1 Moskitt Experiment Setup

The test suite applied on Moskitt has been designed by PRODevelop and consists of 9 test cases, focusing on the generation of scripts from UML diagrams. For this experiment, we apply these 9 tests in different conditions. First we consider a normal execution of Moskitt, i.e. without any monitoring involved. Then we consider an instrumented version of Moskitt where method calls are monitored. Finally, we apply the test cases to an instrumented version of Moskitt where both method calls and variables are monitored.

For this experiment 5 of the Moskitt plugins have been instrumented:

- es.cv.gvcase.mdt.common
- es.cv.gvcase.mdt.db.migration
- es.cv.gvcase.launcher.atl
- es.cv.gvcase.launcher.xpand
- es.cv.gvcase.mdt.db.diagram

## 2.2 Moskitt Overhead induced by Self-Healing

This section presents the results obtained from running test cases on several versions of Moskitt, taking into account monitoring of method calls and variables. These results are presented in Tables 2.1, 2.2, 2.3, and 2.5. These 3 first tables report on experiment conducted respectively with a normal execution of Moskitt (i.e. without any monitoring), with an instrumented version of Moskitt that monitors method calls, and with an instrumented version of Moskitt that monitors both method calls and application variables.
These tables provides some of the metrics usually considered by PRODevelop when testing. These metrics consist of the average execution time, standard deviation, maximum and minimum time as well as the median time.

<table>
<thead>
<tr>
<th>Test Id</th>
<th>Avg Time ms</th>
<th>Std Dev</th>
<th>Max time ms</th>
<th>Min Time ms</th>
<th>Median Time ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>5375</td>
<td>503</td>
<td>5950</td>
<td>4560</td>
<td>5670</td>
</tr>
<tr>
<td>Test2</td>
<td>6636</td>
<td>469</td>
<td>7580</td>
<td>6030</td>
<td>6650</td>
</tr>
<tr>
<td>Test3</td>
<td>6438</td>
<td>919</td>
<td>7800</td>
<td>4730</td>
<td>6370</td>
</tr>
<tr>
<td>Test4</td>
<td>7150</td>
<td>357</td>
<td>7560</td>
<td>6380</td>
<td>7200</td>
</tr>
<tr>
<td>Test5</td>
<td>6536</td>
<td>538</td>
<td>7190</td>
<td>5650</td>
<td>6630</td>
</tr>
<tr>
<td>Test6</td>
<td>6625</td>
<td>588</td>
<td>7270</td>
<td>5650</td>
<td>7030</td>
</tr>
<tr>
<td>Test7</td>
<td>6843</td>
<td>360</td>
<td>7530</td>
<td>6300</td>
<td>6830</td>
</tr>
<tr>
<td>Test8</td>
<td>6695</td>
<td>522</td>
<td>7670</td>
<td>5990</td>
<td>6810</td>
</tr>
<tr>
<td>Test9</td>
<td>6518</td>
<td>784</td>
<td>7650</td>
<td>5140</td>
<td>6950</td>
</tr>
</tbody>
</table>

Table 2.1: Moskitt test result in normal execution, i.e. no supervision applied

<table>
<thead>
<tr>
<th>Test Id</th>
<th>Avg Time ms</th>
<th>Std Dev</th>
<th>Max time ms</th>
<th>Min Time ms</th>
<th>Median Time ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>6571</td>
<td>778</td>
<td>8340</td>
<td>5380</td>
<td>6690</td>
</tr>
<tr>
<td>Test2</td>
<td>7901</td>
<td>761</td>
<td>8580</td>
<td>5890</td>
<td>8150</td>
</tr>
<tr>
<td>Test3</td>
<td>8114</td>
<td>1034</td>
<td>9240</td>
<td>5940</td>
<td>8690</td>
</tr>
<tr>
<td>Test4</td>
<td>8258</td>
<td>665</td>
<td>9340</td>
<td>7110</td>
<td>8270</td>
</tr>
<tr>
<td>Test5</td>
<td>8393</td>
<td>813</td>
<td>10320</td>
<td>7320</td>
<td>8390</td>
</tr>
<tr>
<td>Test6</td>
<td>8487</td>
<td>649</td>
<td>9210</td>
<td>7180</td>
<td>8610</td>
</tr>
<tr>
<td>Test7</td>
<td>8621</td>
<td>652</td>
<td>9370</td>
<td>6890</td>
<td>8880</td>
</tr>
<tr>
<td>Test8</td>
<td>7994</td>
<td>1143</td>
<td>9860</td>
<td>6010</td>
<td>8290</td>
</tr>
<tr>
<td>Test9</td>
<td>8597</td>
<td>575</td>
<td>9410</td>
<td>7550</td>
<td>8770</td>
</tr>
</tbody>
</table>

Table 2.2: Moskitt test results in supervision execution where variables are not monitored.

<table>
<thead>
<tr>
<th>Test Id</th>
<th>Avg Time ms</th>
<th>Std Dev</th>
<th>Max time ms</th>
<th>Min Time ms</th>
<th>Median Time ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>6386</td>
<td>1024</td>
<td>9300</td>
<td>5570</td>
<td>6140</td>
</tr>
<tr>
<td>Test2</td>
<td>7962</td>
<td>1235</td>
<td>9500</td>
<td>5310</td>
<td>8410</td>
</tr>
<tr>
<td>Test3</td>
<td>8232</td>
<td>1059</td>
<td>9770</td>
<td>6460</td>
<td>8820</td>
</tr>
<tr>
<td>Test4</td>
<td>8726</td>
<td>217</td>
<td>9130</td>
<td>8420</td>
<td>8670</td>
</tr>
<tr>
<td>Test5</td>
<td>8460</td>
<td>546</td>
<td>9280</td>
<td>7500</td>
<td>8390</td>
</tr>
<tr>
<td>Test6</td>
<td>8681</td>
<td>725</td>
<td>9720</td>
<td>7310</td>
<td>8970</td>
</tr>
<tr>
<td>Test7</td>
<td>8811</td>
<td>530</td>
<td>9960</td>
<td>8050</td>
<td>8710</td>
</tr>
<tr>
<td>Test8</td>
<td>8193</td>
<td>521</td>
<td>8970</td>
<td>7410</td>
<td>8000</td>
</tr>
<tr>
<td>Test9</td>
<td>8704</td>
<td>495</td>
<td>9480</td>
<td>7940</td>
<td>8770</td>
</tr>
</tbody>
</table>

Table 2.3: Moskitt test results in supervision execution where variables are monitored.
Table 2.4 indicates for each test case the number of events that were monitored. This number represents the sum over the 10 versions of each test case, i.e. the execution of 1 instance of each tests contains about 10 times less events. The interesting value of this table is the overhead per event, i.e. how much extra time supervision induces on average on each method call. The obtained figures vary between 0.003ms and 0.007ms. This can be compared to the figures obtained for the applications of the Dacapo benchmark in Deliverable D6.4. In that deliverable, average overhead per event varied between 0.004ms and 3.16ms. Therefore even when taking variables into account, our approach performs better on Moskitt as on the applications of the Dacapo benchmark.

<table>
<thead>
<tr>
<th>Test</th>
<th>Nb Events Observed without Variables</th>
<th>overhead per Event without variables</th>
<th>Nb Events Observed with Variables</th>
<th>overhead per Event in ms with variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test2</td>
<td>2922914</td>
<td>0.004 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test3</td>
<td>2987774</td>
<td>0.004 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test4</td>
<td>3094568</td>
<td>0.005 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test5</td>
<td>3104135</td>
<td>0.004 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test6</td>
<td>3110244</td>
<td>0.006 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test7</td>
<td>3146686</td>
<td>0.006 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test8</td>
<td>3074279</td>
<td>0.006 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test9</td>
<td>3183522</td>
<td>0.004 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test10</td>
<td>3080305</td>
<td>0.007 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Number of events monitored for each test and overhead per event in ms.

Finally, Table 2.5 summarizes the results presented in the previous tables. It indicates that the overhead induced by monitoring method calls of 5 plugins (listed in Section 2.1) is around 24% and that adding the monitoring of variables for these 5 plugins corresponds to an overhead of about 27%.

<table>
<thead>
<tr>
<th>Normal Mode</th>
<th>Avg Time in ms and percentage overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Mode</td>
<td>6680 ms</td>
</tr>
<tr>
<td>Supervised Application with No Variable</td>
<td>8295 ms</td>
</tr>
<tr>
<td>Percentage Overhead w.r.t Normal Mode</td>
<td>24.18%</td>
</tr>
<tr>
<td>Supervised Application with Variables</td>
<td>8471 ms</td>
</tr>
<tr>
<td>Percentage Overhead w.r.t Normal Mode</td>
<td>26.81%</td>
</tr>
</tbody>
</table>

Table 2.5: Comparison between FastFix overheads induced by the self-healing component in the case where variables are taken into account and in the case where they are not.

These figures can also be compared to the ones obtained from applying our approach to applications from the Dacapo benchmark. In that case, percentage overhead varied between 13% and 272%. Therefore even when taking variables into account, our performance on Moskitt are on the lower range of the ones obtained on the applications of the Dacapo benchmark.
3 Conclusion and Next-Steps
Bibliography


