TOWARDS PROVIDING TEST CASE META-DATA TO ASSIST DYNAMIC REGRESSION

TEST SCHEDULING

A paper
Submitted to the Graduate Faculty
Of the
North Dakota State University
Of Agriculture and Applied Science

By

Arti Katiyar

In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

November 2012

Fargo, North Dakota
The Supervisory Committee certifies that this disquisition complies with North Dakota State University’s regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

DR. KENDALL NYGARD
Chair

DR. TARIQ KING

DR. SIMONE LUDWIG

DR. LIMIN ZHANG

Approved by Department Chair:

DR. BRIAN SLATOR
ABSTRACT

Autonomic software is typically characterized by dynamic adaptation, a self-management process in which the system adds, removes, and replaces its own components at runtime. At the end of maintenance, the modified software would be retested to: (1) validate added or updated software features, and (2) ensure that new errors were not introduced into previously tested components, i.e., regression testing. King et al [14] introduced an implicit self-test characteristic into autonomic software. His approach modifies the success of a runtime testing approach for autonomic software depends on the way tests are selected and scheduled for execution, which still remains an open problem. This paper focuses on determining the information that would be collected during initial test runs to facilitate selecting and prioritizing automated tests. Finally, a test case metadata provider is developed which analyze the generated test case information, which can assist in dynamics regression test scheduling.
ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Kendall Nygard, for his assistance, support and guidance. Dr. King’s recommendations and suggestions have been invaluable for this work.

I would also like to thank Dr. Kendall Nygard, Dr. Simone Ludwig, and Dr. Limin Zhang for being a part of my master’s paper committee.

I would also like to thank Prof. John Nowatzki for giving me funding in his department. Without getting funding in his department I would not have started my Master’s.

Finally, words alone cannot express the thanks I owe to my friends for their continuous support and encouragement.

I would like to express my special thanks to Annaji Sharma Ganti without his support and guidance my journey would never have been competed. Thank You with all my heart and for always being there with me!

Finally, I take this opportunity to express the profound gratitude to my beloved parents, grandparents, and my siblings for their love and continuous support – both spiritually and materially. I also would like to thank my nephew, Devank Katiyar, for the laughter he gifted to me.
# TABLE OF CONTENTS

ABSTRACT ......................................................................................................................... iii

ACKNOWLEDGEMENTS ....................................................................................................... iv

LIST OF TABLES .................................................................................................................. vii

LIST OF FIGURES ............................................................................................................... viii

LIST OF ABBREVIATIONS ................................................................................................... ix

CHAPTER 1. INTRODUCTION ................................................................................................. 1

1.1. Research Problem ........................................................................................................ 2

1.2. Objectives ...................................................................................................................... 2

1.3. Outline of the Paper .................................................................................................... 3

CHAPTER 2. BACKGROUND AND RELATED WORK ............................................................. 4

2.1. Autonomic and Adaptive Computing ........................................................................ 4

2.2. Software Testing .......................................................................................................... 5

2.3. Testing Autonomic Software ..................................................................................... 6

2.3.1. Autonomic Self-Testing ....................................................................................... 6

2.3.2. Test Managers ...................................................................................................... 7

2.3.3. Regression Test Scheduling ................................................................................ 8

2.4. Related Work .............................................................................................................. 9

CHAPTER 3. SURVEY OF FRAMEWORKS AND TOOLS ......................................................... 11

3.1. Search Criteria .......................................................................................................... 11

3.2. Search Results .......................................................................................................... 11

3.2.1. Test Case Performance Tools ............................................................................ 11

3.2.2. Coverage Tools .................................................................................................... 18
3.3. Automated Testing Tools..................................................................................................................22
3.4. Evaluation and Tool Selection...........................................................................................................22

CHAPTER 4. TOWARDS BUILDING A TESTCASE METADATA PROVIDER..............................................24

4.1. Step 1: Setting Up the Environment....................................................................................................26
4.2. Step 2: Identify an Existing Project with Automated Tests.................................................................27
4.3. Step 3: Generate Memory Usage, Execution Time and Code Coverage Metrics...............................27
4.4. Step 4: Design and Develop a Test Case Metadata Provider............................................................30

CHAPTER 5. CONCLUSION AND FUTURE WORK.................................................................................41

REFERENCES............................................................................................................................................43
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Definition of the Execution Time Objects</td>
<td>13</td>
</tr>
<tr>
<td>3.2. Definition of the Memory Objects</td>
<td>14</td>
</tr>
<tr>
<td>3.3. Definition of the ECL Emma Objects</td>
<td>19</td>
</tr>
<tr>
<td>3.4. Types of Coverage Analysis</td>
<td>21</td>
</tr>
<tr>
<td>3.5. Calculation of the Total Coverage Percentage(TPC)</td>
<td>21</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. Closed Loops of Control in Autonomic Manager</td>
<td>5</td>
</tr>
<tr>
<td>3.1. Display of the Execution Time</td>
<td>12</td>
</tr>
<tr>
<td>3.2. Display of the Memory Statistics Time</td>
<td>13</td>
</tr>
<tr>
<td>3.3. Display of the Thread Analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.4. Display of the Performance Test Case</td>
<td>16</td>
</tr>
<tr>
<td>3.5. Display of the Elapsed Time of the Calculator Test Case and the Test Result</td>
<td>17</td>
</tr>
<tr>
<td>3.6. Display of the Test Case Result</td>
<td>17</td>
</tr>
<tr>
<td>3.7. Coverage of the Calculator Class</td>
<td>19</td>
</tr>
<tr>
<td>3.8. Displays the Coverage of the Coffeemaker Test Case</td>
<td>22</td>
</tr>
<tr>
<td>4.1. Steps to Create a Test Case Metadata Provider</td>
<td>26</td>
</tr>
<tr>
<td>4.2. RealEstate Automated Tests Code Coverage Metrics</td>
<td>28</td>
</tr>
<tr>
<td>4.3. RealEstate Automated Tests Memory Usage Metrics</td>
<td>29</td>
</tr>
<tr>
<td>4.4. RealEstate Automated Tests Execution Time Metrics</td>
<td>30</td>
</tr>
<tr>
<td>4.5. XML Report Generated from TPTP Tool</td>
<td>31</td>
</tr>
<tr>
<td>4.6. Serialized XML Report</td>
<td>32</td>
</tr>
<tr>
<td>4.7. Flowchart of the Test Case Metadata Provider</td>
<td>33</td>
</tr>
<tr>
<td>4.8. Class Diagram – Test Case Code Coverage Metadata Provider</td>
<td>34</td>
</tr>
<tr>
<td>4.9. Class Diagram – Test Case Execution Time Metadata Provider</td>
<td>35</td>
</tr>
<tr>
<td>4.10. Class Diagram – Test Case Memory Metadata Provider</td>
<td>36</td>
</tr>
<tr>
<td>4.11. Memory Metrics of the RealEstate Test Cases</td>
<td>39</td>
</tr>
<tr>
<td>4.12. Sorted Memory Metrics of the RealEstate Test Case</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

AC……………Autonomic Computing

AST……………Autonomic Self-Testing

AM……………Autonomic Manager

TPTP…………Test & Performance Tools Platform

TCP……………Total Coverage Percentage

JAAF…………Java self-Adaptive Agent Framework
CHAPTER 1. INTRODUCTION

The autonomic computing (AC) paradigm describes a new generation of software that is capable of self-configuration, self-optimization, self-protection, and self-healing. AC seeks to address the problem of managing highly complex systems through the development of software that automates low-level decisions and tasks, and allows administrators to specify the goals of the system as high-level objectives. Autonomic software is typically characterized by dynamic adaptation, a self-management process in which the system adds, removes, and replaces its own components at runtime. Although AC proposes to automate traditional software maintenance tasks using a highly dynamic self-management infrastructure, its architectural blueprint does not provide support for testing self-adaptive changes made to the software at runtime.

Lack of runtime testing support in autonomic software can have a negative impact on its reliability, since new errors may be introduced into the system during the adaptation process. Researchers have therefore developed approaches for integrating runtime testing into the workflow of autonomic software. King et al. [14] introduced an implicit self-test characteristic into autonomic software. Their approach tailors the existing self-management infrastructure for software testing activities such as test execution, code coverage analysis, and post-test evaluation. Adaptation and testing concerns are kept separate by using independent testing agents that validate the actions of adaptation agents. Da Costa et al. [22] offer an alternative runtime testing approach for self-adaptive software. They modify the autonomic control loop of adaptation agents to include a self-test activity. Each autonomous agent in the software has the combined responsibilities of adaptation and testing. In general, the topic of testing autonomic software at runtime has received little attention in the research literature (Salehie & Tahvildari, [23]).
1.1. Research Problem

The success of a runtime testing approach for autonomic software depends on the way tests are selected and scheduled for execution which still remains an open problem. The runtime testing processes in autonomic software can be divided into three sub problems: (1) selects a subset of tests for validating an adaptive change to the system based on existing regression testing strategies; (2) orders the selected tests for execution in a manner that seeks to maximize one or more predefined testing goals, and (3) revises the established test schedule in response to real-time changes in the autonomic system.

1.2. Objectives

The research concerns the first two sub problems identified for runtime testing processes in autonomic software. The paper is focused towards developing a systematic approach to provide test related information to a dynamic regression test scheduler. The major points covered in this paper include:

1. Identifying the information that should be collected during initial test runs to facilitate selecting and, prioritizing test cases for dynamically scheduling regression tests. Examples include risk and criticality metrics; rates of fault detection and code coverage; and memory, timing, and processing characteristics of tests.

2. Choosing an appropriate set of static and dynamic analysis tools that support automated gathering of test-related information.

3. Developing a test case meta data provider that assists dynamically scheduling of regression tests.
Selecting an existing project with unit tests. Generating test-related information using the tools selected in step 2. Analysing the generated test information using the test meta data provider developed in step 3. Also allowing clients to sort the analysed information.

1.3. Outline of the Paper

The rest of this paper is organized as follows: Chapter 2 provides the background information for Autonomic Systems and also delves into a literature review of self-testing in autonomic systems that this research builds upon. Chapter 3 presents a survey on the tools and plug-in to generate test related information Chapter 4 describes the steps for building a test case meta data provider and also perform an experiment on a project using the developed test case meta data provider. Related work is discussed in Chapter 5 and finally Chapter 6 concludes the paper.
CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter introduces the basics of autonomic computing systems and software testing. It also includes some ideas from other literatures inclined towards self-testing in autonomic systems.

2.1. Autonomic and Adaptive Computing

Autonomic Computing (AC) is a system development paradigm, which stems from the concept of the human autonomic nervous system [10]. The human body has good mechanisms for detecting and repairing anomalies such as physical damage. The autonomic nervous system is responsible for such monitoring, control, and regulation of vital functions without the need for human intervention. Similarly, AC systems can monitor themselves and take action when external changes occur. One of the primary objectives of AC is to address the software complexity problem by making systems easier to manage and integrated [12].

Four commonly cited features of AC systems are [11]:

1. **Self-Configuration** – the ability to dynamically configure and reconfigure itself under changing the conditions, and in accordance with high-level policies representing business-level objectives;
2. **Self-Healing** – the ability to detect failed components and remove or replace them with other components without disrupting the system. In addition, this characteristic may involve the prediction of problems to avoid failures;
3. **Self-Protection** – the ability to identify and detect attacks and cover various aspects of system security at the platform, operating system, and application levels; and
4. **Self-Optimization** – the ability to maximize resource allocation and utilization for satisfying user requests.

Autonomic software must be able to dynamically observe its own structure and behavior (introspection); adapt or modify its own structure and behavior (intercession) and be aware of their
environmental and operational contexts [12]. At the core of AC systems is the concept of an Autonomic Manager (AM). As shown in Figure 2.1 [14], AMs are responsible for implementing closed control loops that monitor, analyze, plan, and execute changes to achieve self-management goals. Sensors are built on managed resources to provide AMs with mechanisms for introspection, while effectors provide intercession mechanisms. During self-management AMs may dynamically add, remove, or replace components of the AC system, a process known as *Dynamic Software Adaptation*.

![Figure 2.1. Closed Loops of Control in Autonomic Manager [14]](image)

2.2. Software Testing

Software testing [13] is the process of operating software under specified conditions, observing or recording the results and making an evaluation of some aspect of the software.
The two main test design methods are Black Box and White Box Testing. Black Box Testing concentrates on testing the functional requirements and does not use knowledge of the internal structure. White box testing is based on covering elements of the program structure, and employs knowledge of the internal workings of the program. Behavioral, functional, opaque, and closed and specification based are the alternative names for black box. Structural, glass box and implementation based are synonyms for white box. The two methods can be applied at unit, integration, system, and regression level of testing [22]. Although black box as well as white box testing is equally essential, using only one is insufficient. So, a combination of black box as well as white box testing called Gray box testing is used in this paper.

Regression testing determines whether modifications to software due to maintenance have introduced new errors into previously tested code. This may involve re-running the entire test suite (retest-all), or selecting a subset of the initial test suite for execution (selective regression testing). Techniques for regression test selection include dataflow, random, safe and test minimization.

2.3. Testing Autonomic Software

Testing plays a vital role in software development. It is not just enough for a software to be functionally excellent, but also necessary to be of high quality, thus enhancing user experience. The concept of autonomic systems with a capability to self-heal and self-protect at runtime have been recently gaining popularity. Many researchers have worked on this idea and developed self-testing frameworks for autonomic systems. By referring to their papers, in this section I have summarized two different approaches that each provides a self-testing framework for autonomic systems.

2.3.1. Autonomic Self-Testing

King et al. [27] proposed a self-testing framework that is capable of dynamically validating the behavior of changed components through regression testing in autonomic computing systems.
This process of validating is based on two key strategies: (1) *Safe Adaptation with Validation* – tests autonomic changes directly on managed resources during the adaptation process, and (2) *Replication with Validation* – tests autonomic changes using copies of managed resources. The Safe Adaptation with Validation strategy must be used only if the process of duplicating managed resources in autonomic computing system is expensive. To support their research investigations, the authors developed a prototype that implements autonomic self-testing according to the proposed validation strategies [26, 28, and 19].

Da Costa et al. [1] have extended the Java self-Adaptive Agent Framework (JAAF) [17] by introducing a new activity called self-test. Self-testing in JAAF is embedded within the control closed loop of collect, analyze, plan and execute components. The self-test activity has ability to validate the new behavior and checks for its adequacy with the new environment before adapting it. The feasibility of the proposed approach is demonstrated by a case study where a system responsible for generating susceptibility maps. The susceptibility maps application makes use of different web-services that are capable of dynamic adaptation. Prior to making an adaptive change, a set of test cases is applied on the given behavior and the execution takes place on the result of pass and fail of test cases. To the best of our knowledge, there are no other approaches in the literature that introduce self-testing into autonomic systems.

### 2.3.2. Test Managers

King et al. [18] Test managers (TMs) extend the concept of autonomic managers to testing activities. Like autonomic managers, TMs may also be Orchestrating or Touchpoint. Orchestrating TMs coordinate high-level testing activities and manage Touchpoint TMs, while Touchpoint TMs perform low-level testing tasks on managed resources. During system execution, Orchestrating AMs will notify Orchestrating TMs that some change request requires validation. Orchestrating TMs will
then coordinate the testing activities necessary to validate that change request using the managed resource or a copy. TMs will be responsible for: To perform the aforementioned duties, TMs will contain components that are consistent with the MAPE structure [3] of autonomic managers. A test monitor will be responsible for polling the managed resource, and/or various components within the self-testing framework, to collect any information relevant to the testing process. A test analyzer will then determine whether or not some testing-related activity needs to be performed such as setting up the test environment, generating new test cases, or conducting a post-test evaluation. A test planner will then generate a plan for the testing activity, and a test executer component will perform the required testing tasks. A test knowledge component will serve as a central repository for test artifacts, and coordinate the interactions between the aforementioned components.

2.3.3. Regression Test Scheduling

In related projects the research is based on scheduling the algorithm using the Average Percentage of Faults Detected is only possible when prior knowledge of faults is available. The other technique is discussed for code based test case prioritization very expensive as its execution is slow. To our knowledge, the described approaches are the only published research directions in the area of runtime testing of autonomic software. The research problem to be studied in the paper will investigate a new scheduling approach for runtime testing processes in autonomic software, which runs a set of test cases and sort the test cases according to the test metrics.

The research plan to be presented in the chapter can be summarized as:

Identify the information that should be collected during initial test runs to facilitate selecting, prioritizing, and dynamically scheduling regression tests. Examples include risk and criticality metrics; rates of fault detection and code coverage; and memory, timing, and processing
characteristics of tests. To support automated gathering of test-related information, this step involves choosing an appropriate set of static and dynamic analysis tools.

2.4. Related Work

Researchers have developed approaches for integrating runtime testing into the workflow of autonomic software (King et al., [19]; Da Costa et al., [22]). King et al. [14] introduced an implicit self-test characteristic into autonomic software. Their approach tailors the existing self-management infrastructure for software testing activities such as test execution, code coverage analysis, and post-test evaluation. Adaptation and testing concerns are kept separate by using independent testing agents that validate the actions of adaptation agents. Da Costa et al. (2010) offer an alternative runtime testing approach for self-adaptive software. They modify the autonomic control loop of adaptation agents to include a self-test activity.

Srivastava et al. [15] proposed a prioritization technique and also used a metric called APFD (Average Percentage of Faults Detected) for calculating the effectiveness of the test case prioritization methods. The disadvantage of the method proposed is that calculation of APFD is only possible when prior knowledge of faults is available. APFD calculations therefore are only used for evaluation of effectiveness of various prioritization techniques.

Rothermel et al. [16, 17, and 18] presented 21 different techniques for code based test case prioritization, which are classified into three different groups i.e. comparator group, statement level group and function level group. To measure the effectiveness of these techniques, an experiment was conducted where 7 different programs were taken. Here several dimensions like granularity were taken for test case prioritization. The main disadvantages of code based test case prioritization are it is very expensive as its execution is slow because of the execution of the actual code and code based
test case prioritization may not be sensitive to the correct or incorrect information provided by the testers or the developer.

Srikanth et al. [21] presented a technique that extend the code-coverage TCP techniques and apply test case prioritization at a system-level for both new and regression tests. Here the advantage is that the author uses a system level test case prioritization techniques which is called the Prioritization.
CHAPTER 3. SURVEY OF FRAMEWORKS AND TOOLS

This chapter gives an overview of different tools and plugins and their criteria for selection in order to generate test related information.

3.1. Search Criteria

A survey of tools and plugins was conducted to generate test related information. In order to build an test case meta data provider, tools and plugins should supports the following aspects: (1) easily integrate into a IDE; (2) ability to display generated metrics; (3) Export generated results to a file say xml;

3.2. Search Results

The search led to three categories of tools and frameworks to support the development of a test case meta data provider. These were: (1) Test Case Performance Tools; (2) Coverage Tools and (3) Automated Testing Tools. This subsection provides an overview of some of the tools and plugins found in each category.

3.2.1. Test Case Performance Tools

Performance testing is testing that is performed, to determine how fast some aspect of a system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Any software should be tested for its performance. Following is the list of tools which were used to get the performance metrics of the test cases:

3.2.1.1. Eclipse Tptp

Eclipse Test and Performance Tools Platform (TPTP) Project provides an open platform supplying powerful frameworks and services that allow software developers to build unique test and
performance tools, both open source and commercial, that can be easily integrated with the platform and with other tools.

TPTP addresses the entire test and performance life cycle, from early testing to production application monitoring, including test editing and execution, monitoring, tracing and profiling, and log analysis capabilities.

3.2.1.1.1. Execution Statistics view

Figure 3.1 displays the execution statistics about the application. It provides data such as the number of methods called, and the amount of time taken to execute every method. Execution statistics are available at the package, class, method and instance level. Below is the screenshot:

![Execution Statistics](image)

**Figure 3.1. Display of the Execution Time**

Statistics are displayed for each object type. The data displayed depends what object type is selected. The statistics available for each object are summarized in Table 3.1.
Table 3.1. Definition of the Execution Time Objects

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Time</td>
<td>For any invocation, the base time is the time taken to execute the invocation, excluding the time spent in other methods that were called during the invocation.</td>
</tr>
<tr>
<td>Average Base Time</td>
<td>Base time divided by the number of calls.</td>
</tr>
<tr>
<td>Inherited Base Time</td>
<td>Similar to the base time spent in the selected package or class including the time spent in other inherited methods that were called during the invocation.</td>
</tr>
<tr>
<td>Cumulative Time</td>
<td>For any invocation, the cumulative time is the time taken to execute all methods called from an invocation. If an invocation has no additional method calls, then the cumulative time will be equal to the base time.</td>
</tr>
<tr>
<td>Inherited Cumulative Time</td>
<td>Similar to the cumulative time of the selected package or class including the time spent in other inherited methods that were called during the invocation.</td>
</tr>
<tr>
<td>Calls</td>
<td>The number of calls on the selected method.</td>
</tr>
<tr>
<td>Inherited Calls</td>
<td>The number of calls on the selected method and its inherited methods.</td>
</tr>
</tbody>
</table>

3.2.1.1.2. Memory Statistics view

The Memory Statistics views displays statistics about the application heap. It provides detailed information such as the number of classes loaded, the number of instances that are alive, and the memory size allocated by every class. Figure 3.2 displays Memory statistics that are available at the package, class, and instance level.

Figure 3.2. Display of the Memory Statistics Time

Statistics displayed for each object type. The data displayed depends what object type is selected. The statistics available for each object are summarized in Table 3.2.
Table 3.2. Definition of the Memory Objects

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Instances</td>
<td>The total number of instances that had been created of the selected package, class, or method.</td>
</tr>
<tr>
<td>Live Instances</td>
<td>The number of instances of the selected package, class, or method, where no garbage collection has taken place.</td>
</tr>
<tr>
<td>Collected</td>
<td>The number of instances of the selected package, class, or method that were removed during garbage collection.</td>
</tr>
<tr>
<td>Total Size</td>
<td>The total size (in bytes) of the selected package, class, or method, of all instances that were created for it, including whatever has been removed through garbage collection.</td>
</tr>
<tr>
<td>Active Size</td>
<td>The summed size of all live instances.</td>
</tr>
</tbody>
</table>

3.2.1.3. Thread Analysis view

Figure 3.3 displays the Thread Analysis statistics about the thread count, state, and interactions. It provides detailed information such as the number of classes and threads loaded, the number of instances that are alive, and the thread interaction for every class. Figure 3.3 displays the Thread analysis statistics that are available at the class level.

![Figure 3.3. Display of the Thread Analysis](image)

Statistics displayed for each object type. The data displayed depends what object type is selected. The statistics available for each object are summarized in the table below. **State**: The state of the listed threads (running, sleeping, waiting, blocked).

3.2.1.2. JUnitPerf

JUnitPerf is a collection of JUnit test decorators used to measure the performance and scalability of functionality contained within existing JUnit tests. JUnitPerf contains the following JUnit test decorators:
A TimedTest is a JUnit test decorator that measures the total elapsed time of a JUnit test and fails if the maximum time allowed is exceeded. A timed test tests time-critical code, such as a sort or search.

A TimedTest is constructed with an instance of a JUnit test, along with the maximum allowed execution time in milliseconds. Figure 3.4 shows an example of a timed test that fails if the elapsed time of CalcJUnitTest.testSum() method exceeds two seconds:

```java
public class ExampleTimedTest {

    public static Test suite() {
        long maxElapsedTime = 2000;

        Test testCase = new CalcJUnitTest("testSum");

        Test timedTest = new TimedTest(testCase, maxElapsedTime);

        return timedTest;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```
In the example above, the total elapsed time is checked once the method under test completes. If the total time exceeds two seconds, the test fails.

Another option is for the test to fail immediately if the maximum allowed execution time is exceeded. Figure 3.5 shows an example of a timed test that causes immediate failure:

```java
public class ExampleTimedTest {
    public static Test suite() {
        long maxElapsedTime = 01;
        Test testCase = new CalcJUnitTest("testSum");
        Test timedTest = new TimedTest(testCase, maxElapsedTime);
        return timedTest;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Figure 3.4. Display of the Performance Test Case
Figure 3.5. Display of the Elapsed Time of the Calculator Test Case and the Test Result

Time: 0.003
There was 1 failure:
1) testSum(CalcJUnitTest)junit.framework.AssertionFailedError: Maximum elapsed time exceeded! Expected 1ms, but was 2ms.
   at com.clarkware.junitperf.TimedTest.runUntilTestCompletion(TimedTest.java:161)
   at com.clarkware.junitperf.TimedTest.run(TimedTest.java:138)
   at ExampleTimedTest.main(ExampleTimedTest.java:16)
FAILURES!!!
Tests run: 1, Failures: 1, Errors: 0
BUILD SUCCESSFUL (total time: 1 second)

Figure 3.6. Display of the Test Case Result

Figure 3.6 displays the result of the logging statement after the immediate failure.
3.2.2. Coverage Tools

Coverage is a quality assurance metric which determines how thoroughly a test suite exercises a given program [1]. Coverage-based testing can be applied to any stage of testing including unit, integration or system testing. The output of coverage measurement can be used in several ways to improve the verification process. The coverage tool can detect illegal events that occur and help find bugs in the design. It can help to find holes in the testing, i.e. areas that are not covered [2]. Test coverage also helps in Regression testing, test case prioritization, test suite augmentation and test suite minimization [3]. Below are the coverage tools that I did the survey on to figure out the dynamic scheduling of the test case:

3.2.2.1. ECL Emma

It is an open-source tool for measuring and reporting code coverage for Java [5]. It can instrument classes for coverage either offline (before they are loaded) or on the fly (using an instrumenting application class loader). Supported coverage types are class, method, line and basic block. [3]. Figure 3.7 shows the code coverage for the calculator class.

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.
3.2.2.1.1. Counters

When analyzing the coverage data collected during a coverage session different counters are evaluated for each Java element. Table 3.3 defines the ECL Emma objects. The coverage view allows selecting from its drop-down menu which counter should be displayed:

Table 3.3. Definition of the ECL Emma Objects

<table>
<thead>
<tr>
<th>Basic Blocks</th>
<th>This is EMMA's fundamental unit of coverage. A basic block is a sequence of bytecode instructions without any jumps or jump targets. A basic block is considered as covered when its last instruction has been executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>The number of covered Java source code lines. A source line is considered as covered if it contains at least one covered basic block. This counter is only available if the line numbers are included in the compiled class files (debug mode).</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>The number of bytecode instructions within the basic blocks. If no line information is available this is a good approximation for line coverage.</td>
</tr>
<tr>
<td>Methods</td>
<td>A method is considered as covered if at least one basic block of the method has been executed.</td>
</tr>
<tr>
<td>Types</td>
<td>A Java type is considered as covered if it has been loaded and initialized.</td>
</tr>
</tbody>
</table>
3.2.2.2. **Cobertura**

Cobertura [4] is a free Java tool that calculates the percentage of code accessed by tests. It is used to identify which parts of your Java program are lacking test coverage. It is based on coverage. *Some of the features supported by Cobertura are:* Can be executed from ant or from the command line; Instruments Java byte code after it has been compiled; Can generate reports in HTML or XML; Shows the percentage of lines and branches covered for each class, each package and for the overall project; Shows the McCabe cyclomatic code complexity of each class, and the average cyclomatic complexity for each package, and for the overall product; Can sort HTML results by class name, percent of lines covered, percent of branches covered, etc. and can sort in ascending or descending order.

3.2.2.3. **Code Cover**

Code cover [6] is a free testing tool for Java programmers. It is fully integrated into Eclipse and performs source instrumentation for coverage measurement especially for condition coverage. It helps to increase test quality.

3.2.2.4. **Clover**

Clover is designed to measure code coverage in a way that fits seamlessly with your current development environment and practices, whatever they may be. Clover's IDE Plugins provide developers with a way to quickly measure code coverage without having to leave the IDE. Clover's Ant and Maven integrations allow coverage measurement to be performed in Automated Build and Continuous Integration systems, and reports generated to be shared by the team. Figure 3.8 displays the coverage of the CoffeeMaker test case.
3.2.2.5. Types of Coverage measured

Table 3.4 shows the Clover measures three basic types of coverage analysis:

Table 3.4. Types of Coverage Analysis

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Statement coverage measures whether each statement is executed.</td>
</tr>
<tr>
<td>Branch</td>
<td>Branch coverage (sometimes called Decision Coverage) measures which possible branches in flow control structures are followed. Clover does this by recording if the boolean expression in the control structure evaluated to both true and false during execution.</td>
</tr>
<tr>
<td>Method</td>
<td>Method coverage measures if a method was entered at all during execution.</td>
</tr>
</tbody>
</table>

Clover uses these measurements to produce a Total Coverage Percentage for each class, file, and package and for the project as a whole. The Total Coverage Percentage allows entities to be ranked in reports. Table 3.5 shows the calculation of Total Coverage Percentage (TPC).

Table 3.5. Calculation of the Total Coverage Percentage (TPC)

\[
TPC = \frac{BT + BF + SC + MC}{2B + S + M}
\]

where

- BT - branches that evaluated to "true" at least once
- BF - branches that evaluated to "false" at least once
- SC - statements covered
- MC - methods entered

- B - total number of branches
- S - total number of statements
- M - total number of methods
3.3. Automated Testing Tools

JUnit [8] is a simple Java testing framework to write tests for a Java application. JUnit is linked as a Java Archive (JAR) at compile-time; the framework resides under packages junit.framework for JUnit 3.8 and earlier and under org.junit for JUnit 4 and later [9]. Some of the features supported by JUnit are: assertions for testing expected results; test fixtures for sharing common test data; test suites for easily organizing and running tests; and graphical and textual test runners. JUnit was built in Net beans IDE and Eclipse IDE for creating test cases and running test suites.

3.4. Evaluation and Tool Selection

The purpose of conducting the survey was to discover frameworks and tools that could be used to help us developing software for generating test case related information. Tools were selected based on their cohesiveness with respect to programming language, development environment, and interoperability. Several open-source testing tools are available for Java and so this was the
programming language of choice. The Eclipse Test and Performance Tools Platform (TPTP) Project provides an open platform supplying powerful frameworks is available as a library for Java, and is therefore easy to integrate with JUnit and other testing tools. Furthermore, Code Clover provides the code coverage for the test cases. The described tools were therefore preferred for building the software for generating test case related information.
CHAPTER 4. TOWARDS BUILDING A TESTCASE METADATA PROVIDER

As discussed in Chapter 1 the research problem and the solution for the problem. In this chapter I will explain in detail why this research was necessary and how the current work will be useful for larger framework.

According to the problems that are discussed in Chapter 1 the current research shows the development of the two problems. This chapter describes in detail the solution and the development of the two problems.

The third research problem discussed in Chapter 1 “revises the established test schedule in response to real-time changes in the autonomic system.” To develop the solution for this problem the current research work was required. The problem concerns the need for testing the real-time changes in the operation of the autonomic software. Recall that testing must occur online, while the system is executing. As such, the test scheduler must be capable of dynamically scheduling or rescheduling individual test cases based on the test metrics, or even suspending the entire testing process so that (non-testing) system goals can be met. For example, if a user makes an explicit request for a resource that is currently locked by the testing process, scheduled tests that lock that resource may be postponed so that availability constraints would not be violated. In some cases, administrators may decide that validating an adaptive change (for example to a critical component) should take precedence over the prevention of a constraint violation. The scheduling solution must therefore be highly flexible and reactive with respect to the ordering and execution of tests.

The solutions for three problems are:

1. Identify the information that should be collected during initial test runs to facilitate selecting, prioritizing, and dynamically scheduling regression tests. Examples include risk and criticality metrics; rates of fault detection and code coverage; and memory, timing, and
processing characteristics of tests. To support automated gathering of test-related information, this step involves choosing an appropriate set of static and dynamic analysis tools.

2. Formulate an approach that uses the information collected at Step 1, along with a predefined testing metrics, to dynamically schedule regression tests in autonomic software. The testing metrics should specify the test selection strategies, testing goals to be maximized, and event-action pairs for responding to real-time events. We position the approach according to the dynamic scheduling categories and techniques by Ouelhadj & Petrovic (2009).

3. Implement the approach formulated at Step 2 in an autonomic way, and perform experiments to evaluate its practicality, benefits, and disadvantages when compared with static/naive approaches. We use the integrated self-testing framework and it’s supporting object-oriented design for the implementation (King et al., 2007; King et al., 2008). However, we plan to extend the existing designs with safety mechanisms to facilitate suspending and resuming the testing process as part of dynamic scheduling. [25]

The current research has a solution for the first two problems (1) selects a subset of tests for validating an adaptive change to the system based on existing regression testing strategies; and (2) orders the selected tests for execution in a manner that seeks to maximize one or more predefined testing goals. To develop a third problem “revises the established test schedule in response to real-time changes in the autonomic system” TestCase Metadata provider will be used as a large framework.
This chapter describes the steps taken to generate the Test Case Metadata. Figure 4.1 highlights the various phases of the Test Case Metadata generation process to be carried out. In the following sections each phase is discussed in detail.

4.1. Step 1: Setting Up the Environment

The first step of the process was to install and configure the identified development and test tools from Chapter 3. The Eclipse Classic IDE was installed and configured. Later the eclipse plugins TPTP and Code Cover were also downloaded and configured.
4.2. Step 2: Identify an Existing Project with Automated Tests

The second step of the process was to identify one or more existing java projects that can be easily imported into eclipse IDE and can be easily run with minor modifications. One of the criteria for choosing the project was having a large number of test cases. The Higher the number of test cases in the project the better it is for the prototype. The test cases for the project should be written in JUnit 4.0 or later to work well with the test case analysis tools that are identified in Chapter 3. Apart from this the other criteria’s include the number of number of classes, total number of lines, and number of methods in each class and complexity of the application. All this which would contribute to better analyze the effectiveness of the test case Meta data provided to be built.

A Java project which meets all the above mentioned criteria called RealEstate was chosen for the prototype from the Repository for Open Software Education [24]. The RealEstate example was created by two graduate students at North Carolina State University to be used as a teaching example in the undergraduate software engineering class. The premis of RealEstate is to provide a teaching example of a common board game that most students will be familiar with. The project comes with less than 1000 lines of code which makes it easy to read and understand. Also included are the

4.3. Step 3: Generate Memory Usage, Execution Time and Code Coverage Metrics

The RealEstate project identified in the previous section is imported into eclipse. The eclipse plugin Code Cover is used to generate the Code Coverage metrics (Figure 4.2) for automated test cases. The generated code coverage metrics is then exported to an xml file, RECodeCoverageMetrics.xml
Figure 4.2. RealEstate Automated Tests Code Coverage Metrics

The eclipse plugin TPTP is used to generate the memory usage metrics (Figure 4.3) for automated test cases of the project. The generated memory usage metrics is then exported to an xml file, RE_MemoryMetrics.xml
Figure 4.3. RealEstate Automated Tests Memory Usage Metrics

The eclipse plugin TPTP is used to generate the execution time metrics (Figure 4.4) for automated test cases of the project. The generated execution time metrics is then exported to an xml file, RE_ExecutionMetrics.xml.
4.4. Step 4: Design and Develop a Test Case Metadata Provider

The final step of the process was to develop an application that would generate the test case metadata information reading the generated metrics files from previous step. As the first step in the application development process the requirements of the test case metadata provides were identified as below:

1. The application should be able to read the generated Code Coverage, Memory Usage and Execution Time Metrics xml files.
2. The application should be able to generated analyze the information from the metrics file and store the test case Meta data information in memory.
3. The application should be able to sort the generated test case metadata information depending on the user sort criteria.

4. The application should be able to serialize the generated test case metadata information to xml file.

Using the above requirements a flow chart is developed which represents the control flow of the application to be developed. The application takes one of the metrics file and the sort criteria.

Figure 4.5 displays the XML report that is created by the Eclipse TPTP tool that was discussed in Chapter 3 in detail. This XML provides the information such as package name, class name and memory of each class.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <data Avg_Age="0" Total_Size_bytes="72" Total_Instances="3" Active_Size_bytes="72">
    Live_Instances="3" Package="edu.ncsu.realestate" Class_Name="GoToJailCardTest"/>
  <data Avg_Age="0" Total_Size_bytes="16" Total_Instances="1" Active_Size_bytes="16">
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="TradeDealTest"/>
  <data Avg_Age="0" Total_Size_bytes="16" Total_Instances="1" Active_Size_bytes="16">
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="PropertyCellTest"/>
  <data Avg_Age="0" Total_Size_bytes="144" Total_Instances="6" Active_Size_bytes="144">
    Live_Instances="6" Package="edu.ncsu.realestate" Class_Name="GameboardTest"/>
  <data Avg_Age="0" Total_Size_bytes="112" Total_Instances="7" Active_Size_bytes="112">
    Live_Instances="7" Package="edu.ncsu.realestate" Class_Name="GameMasterTest"/>
  <data Avg_Age="0" Total_Size_bytes="32" Total_Instances="2" Active_Size_bytes="32">
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="CellInfoFormatterTest"/>
  <data Avg_Age="0" Total_Size_bytes="24" Total_Instances="1" Active_Size_bytes="24">
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="CardTest"/>
  <data Avg_Age="0" Total_Size_bytes="48" Total_Instances="2" Active_Size_bytes="48">
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="GainMoneyCardTest"/>
  <data Avg_Age="0" Total_Size_bytes="64" Total_Instances="4" Active_Size_bytes="64">
    Live_Instances="4" Package="edu.ncsu.realestate" Class_Name="UtilityCellTest"/>
  <data Avg_Age="0" Total_Size_bytes="32" Total_Instances="2" Active_Size_bytes="32">
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="PlayerTest"/>
</View>
```

**Figure 4.5. XML Report Generated from TPTP Tool**

The application then analyzes the metrics file and generates the test case metadata information. The above XML report that was generated using the tool is parsed to gather the metrics.
The generated file is then sorted based on the search criteria passed. Figure 4.6 displays the information and serialized xml file. Below is the serialized XML:

```
  <data Avg._Age="0" Total_Size_.bytes="16" Total_Instances="1" Active_Size_.bytes="16"
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="TradeDealTest"/>
  <data Avg._Age="0" Total_Size_.bytes="16" Total_Instances="1" Active_Size_.bytes="16"
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="PropertyCellTest"/>
  <data Avg._Age="0" Total_Size_.bytes="24" Total_Instances="1" Active_Size_.bytes="24"
    Live_Instances="1" Package="edu.ncsu.realestate" Class_Name="CardTest"/>
  <data Avg._Age="0" Total_Size_.bytes="32" Total_Instances="2" Active_Size_.bytes="32"
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="CellInfoFormatterTest"/>
  <data Avg._Age="0" Total_Size_.bytes="32" Total_Instances="2" Active_Size_.bytes="32"
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="PlayerTest"/>
  <data Avg._Age="0" Total_Size_.bytes="48" Total_Instances="2" Active_Size_.bytes="48"
    Live_Instances="2" Package="edu.ncsu.realestate" Class_Name="GainMoneyCardTest"/>
  <data Avg._Age="0" Total_Size_.bytes="64" Total_Instances="4" Active_Size_.bytes="64"
    Live_Instances="4" Package="edu.ncsu.realestate" Class_Name="UtilityCellTest"/>
  <data Avg._Age="0" Total_Size_.bytes="72" Total_Instances="3" Active_Size_.bytes="72"
    Live_Instances="3" Package="edu.ncsu.realestate" Class_Name="GoToJailCardTest"/>
  <data Avg._Age="0" Total_Size_.bytes="112" Total_Instances="7" Active_Size_.bytes="112"
    Live_Instances="7" Package="edu.ncsu.realestate" Class_Name="GameMasterTest"/>
  <data Avg._Age="0" Total_Size_.bytes="144" Total_Instances="6" Active_Size_.bytes="144"
    Live_Instances="6" Package="edu.ncsu.realestate" Class_Name="GameboardTest"/>
</View>
```

**Figure 4.6. Serialized XML Report**

In the above XML, all the classes are sorted according to the memory size.
Below static models using UML 2.0 were designed to represent a solution to satisfy the above requirements and the flow of the application.
Figure 4.8 describes the CodeCoverage design.

**FileTypesClass** stores the metrics XML reports that were generated using the reports. The files that were stored are CodeCoverage File, MemoryMetricsFile and ExecutionTimeMetricsFile.

**TestCaseInfoBaseClass** stores the PackageName and ClassName.

**ParserBaseClass** parses **TestCaseInfoBase**.

**TestCaseCodeCoverageMetricsInfoClass** extends **TestCaseInfoBase** stores the attributes of Codecoverage metrics percentage covered methods and statements.

**CodeCoverageMetricsParserClass** extends **ParserBase** to parse the information from **TestCaseCodeCoverageMetricsInfoClass**.

**CoveredMethodComparerClass** compares the memory against each test case to sort them.
SerializerClass is to generate a sorted XML file.

Figure 4.9. Class Diagram – Test Case Execution Time Metadata Provider

Figure 4.9 describes the ExecutionTime design.

FileTypesClass stores the metrics XML reports that were generated using the reports. The files that were stored are CodeCoverage File, MemoryMetricsFile and ExecutionTimeMetricsFile.

TestCaseInfoBase Class stores the PackageName and ClassName.

TestCaseExecutionTimeMetricsInfo Class extends the TestCaseInfoBase and stores the base time and cumulative time of the test case.

ParserBaseClass parses TestCaseInfoBase.

ExecutionTimeParserClass extends ParserBase to parse the information from TestCaseExecutionTimeMetricsInfo.
**BaseTimeComparerClass** compares the base time and cumulative time against each test case to sort them.

**SerializerClass** is to generate a sorted XML file.

![Class Diagram](image)

**Figure 4.10. Class Diagram – Test Case Memory Metadata Provider**

Figure 4.10 describes the Memory metrics design.

**FileTypesClass** stores the metrics XML reports that were generated using the reports. The files that were stored are CodeCoverage File, MemoryMetricsFile and ExecutionTimeMetricsFile.

**TestCaseInfoBaseClass** stores the PackageName and ClassName.
ParserBaseClass parses TestCaseInfoBase.

TestCaseMemoryMetricsInfoClass extends TestCaseInfoBase stores the attributes of Memory metrics LiveInstances, ActiveSize, TotalInstances and TotalSize.

MemoryMetricsParserClass extends ParserBase to parse the information from TestCaseMemoryMetricsInfo.

ActiveSizeComparerClass is to compare against the attributes of Memory metrics LiveInstances, ActiveSize, TotalInstances and TotalSize and to sort them.

SerializerClass is to generate a sorted XML file.

A java application was developed on the basis of the above class diagram using eclipse IDE. The sample code represents a basic test where the test reads the file “RE_MemoryMetrics.xml”, reads and analyzes the test case memory metrics information for the Real Estate project. The unsorted information is serialized to xml and also displays the information to an output window. Then the memory metrics information is sorted, depending on the ActiveSize of the test case. The sorted information is serialized to xml and finally displayed in an output window.

TestCaseMetaDataProvider<TestCaseMemoryMetricsInfo> provider = new TestCaseMetaDataProvider<TestCaseMemoryMetricsInfo>();
ArrayList<TestCaseMemoryMetricsInfo> testcaseinfos = provider.ProvideMetaData("RE_MemoryMetrics.xml", FileTypes.MemoryMetricsFile, "edu.ncsu.realestate");
Serializer<TestCaseMemoryMetricsInfo> generator = new Serializer<TestCaseMemoryMetricsInfo>();
DisplayTestcaseMemoryMetricsInfos(testcaseinfos);
generator.SerializeArrayList("TestCaseMemoryMetricsInfos_BeforeSort.xml", testcaseinfos);
System.out.println("After Sorting");

ArrayList<TestCaseMemoryMetricsInfo> temp = testcaseinfos;
Collections.sort(temp, new ActiveSizeComparer());

DisplayTestCaseMemoryMetricsInfos(temp);

generator.SerializeArrayList("TestCaseMemoryMetricsInfos_AfterSort.xml", temp);

The above sample test generates the information in the output window as shown in Figure 4.11.

On running the above sample test generates the information in the output window as shown in Figure 4.11 and also serializes the information to TestCaseMemoryMetricsInfos_BeforeSort.xml file. Also shown in Figure 4.12 is the sorted information generated in the output window. The code finally serializes the information to the file TestCaseMemoryMetricsInfos_AfterSort.xml.
Figure 4.11. Memory Metrics of the RealEstate Test Cases
Figure 4.12. Sorted Memory Metrics of the RealEstate Test Cases

To summarize, in this chapter we have devised an approach to develop a test case meta data provider that can be integrated into dynamic runtime test scheduler of autonomic software.
CHAPTER 5. CONCLUSION AND FUTURE WORK

The runtime testing processes in autonomic software are divided into three sub problems: (1) select a subset of tests for validating an adaptive change to the system based on existing regression testing strategies; and (2) order the selected tests for execution in a manner that seeks to maximize one or more predefined testing goals, and (3) revise the established test schedule in response to real-time changes in the autonomic system. The paper targets the first two sub-problems leading to assistance for providing the solution for the final problem of scheduling the testing in autonomic systems.

As a preliminary step the test case related information that would assist the runtime test scheduler in selecting tests. The information that needs to be collected during the initial test runs are identified, and include code coverage, memory consumption and execution time associated to a test case. A survey of existing static and dynamics analysis tools that can be used to automate the gathering of test case related information. “Performance testing is testing that is performed, to determine how fast some aspect of a system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage.” According to the definition of performance testing, was looking for the tool which gives better performance, covers the maximum number of test and gives the exact metrics too work better on the research conducted. The Eclipse Test and Performance Tools Platform (TPTP) were selected to retrieve memory usage and execution time information of a test case. In Chapter 3 Section 3.2.1.1 Eclipse Ttp shows the screenshots of the performance of each test case. “Coverage is a quality assurance metric which determines how thoroughly a test suite exercises a given program.” Code Clover was selected to retrieve percent of code coverage by a test case. In Chapter 3
Section 3.2.2.4 Code Clover shows the screenshots of the code coverage which give the detail coverage of each class, methods and average complexity of the class.

A test case meta data provider was developed which not only act as a test case related information provider to the runtime test scheduler but also allow arranging of the test cases depending on the runtime test scheduler testing policy. An experiment was conducted on a sample project, RealEstate, to determine the workings of identified static and dynamics analysis tools in combination with the developed test case meta data provider.

As part of future work, the test case provider needs to be integrated into a regression test scheduler for autonomic software. An approach needs to be developed that uses the information provided by the test case meta data provider, along with a predefined testing policy, to dynamically schedule regression tests in autonomic software. The testing policy should specify the test selection strategies, testing goals to be maximized, and event-action pairs for responding to real-time events. The approach would be positioned according to the dynamic scheduling categories and techniques by Ouelhadj & Petrovic [20]. Implementing the approach formulated in an autonomic, environment and performing experiments would evaluate its practicality, benefits, and disadvantages when compared with static/naive approaches. King et al. [19] integrated self-testing framework and its supporting object-oriented design for the implementation can be used to achieve this. However, it would be desirable to extend the existing designs with safety mechanisms to facilitate suspending and resuming the testing process as part of dynamic scheduling.
REFERENCES


