NOTE TO USERS

This reproduction is the best copy available.

UMI
A SELF-TESTING APPROACH FOR AUTONOMIC SOFTWARE

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Tariq M. King

2009
To: Dean Amir Mirmiran  
College of Engineering and Computing  

This dissertation, written by Tariq M. King, and entitled A Self-Testing Approach for Autonomic Software, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

[Signatures]

Masoud Milani  
Xudong He  
S. Masoud Sadjadi  
Martha Centeno

Peter J. Clarke, Major Professor

Date of Defense: July 10, 2009

The dissertation of Tariq M. King is approved.

[Signature]

Dean Amir Mirmiran  
College of Engineering and Computing

[Signature]

Dean George Walker  
University Graduate School

Florida International University, 2009
DEDICATION

I dedicate this dissertation to Jesus Christ, the one through whom all things have been made, because without him nothing was made that has been made... [John 1:3]
ACKNOWLEDGMENTS

First and foremost I would like to thank my family: Suzanne King, Roosevelt King, Aisha King, Samaria King, Carson King, David Franklin, Patricia Brathwaite, John Matthews, and Christine Rocheford for their loving support throughout my academic journey. My mother Suzanne has always been a firm believer of excellence in education, and this dissertation is a testament to her success as a parent. I extend special thanks to David for introducing me to computing at the young age of five, and providing me with the discipline and focus necessary to reach the top of my field.

I would also like to thank my advisor and friend, Dr. Peter J. Clarke, and my committee members Dr. Masoud Milani, Dr. Xudong He, Dr. S. Masoud Sadjadi, and Dr. Martha Centeno for their guidance and insights on my research. Dr. Clarke is without a doubt the best graduate advisor at FIU, and I will be sure to follow his example by caring for my students and paying forward the many kindnesses he has shown to me. Drs. Sadjadi and Milani have been instrumental to my research through the provision of summer courses, and research programs on autonomic computing.

Lastly, I would like to thank my friends and fellow researchers: Yanelis Hernandez, Jose Morales, Satish Harrylall, Hasina Patel, Tonia Blades, Stacy Johnson, Davina Bush, Djuradj Babich, Jonatan Alava, Alain Ramirez, Rodolfo Cruz, Jairo Pava, Andrew Allen, and Yali Wu. Yanelis and her grandmother Mima have been an amazing support system over the last two and a half years, and this achievement is as much theirs as it is mine. Thank you Yane for being my everything, especially during my back injury, and for introducing me to Dr. Daniel Bravo and his talented staff.
ABSTRACT OF THE DISSERTATION

A SELF-TESTING APPROACH FOR AUTONOMIC SOFTWARE

by

Tariq M. King

Florida International University, 2009

Miami, Florida

Professor Peter J. Clarke, Major Professor

As researchers and practitioners move towards a vision of software systems that configure, optimize, protect, and heal themselves, they must also consider the implications of such self-management activities on software reliability. Autonomic computing (AC) describes a new generation of software systems that are characterized by dynamically adaptive self-management features. During dynamic adaptation, autonomic systems modify their own structure and/or behavior in response to environmental changes. Adaptation can result in new system configurations and capabilities, which need to be validated at runtime to prevent costly system failures. However, although the pioneers of AC recognize that validating autonomic systems is critical to the success of the paradigm, the architectural blueprint for AC does not provide a workflow or supporting design models for runtime testing.

This dissertation presents a novel approach for seamlessly integrating runtime testing into autonomic software. The approach introduces an implicit self-test feature into autonomic software by tailoring the existing self-management infrastructure to runtime testing. Autonomic self-testing facilitates activities such as test execution, code coverage analysis, timed test performance, and post-test evaluation. In addition, the approach is supported by automated testing tools, and a detailed design methodology. A case study that incorporates self-testing into three autonomic applications is also presented. The findings of the study reveal that autonomic self-testing provides a flexible approach for building safe, reliable autonomic software, while limiting the development and performance overhead through software reuse.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Goal and Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Proposed Solution</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Summary of Contributions</td>
<td>10</td>
</tr>
<tr>
<td>1.6 Scope and Limitations</td>
<td>10</td>
</tr>
<tr>
<td>1.7 Outline of the Dissertation</td>
<td>11</td>
</tr>
<tr>
<td>2 LITERATURE REVIEW</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Background</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1 Autonomic Computing</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 Software Testing</td>
<td>18</td>
</tr>
<tr>
<td>2.1.3 Software Design</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>22</td>
</tr>
<tr>
<td>2.2.1 Self-Testable Software</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2 Testing Adaptive Systems</td>
<td>24</td>
</tr>
<tr>
<td>2.2.3 Model Checking Adaptive Systems</td>
<td>26</td>
</tr>
<tr>
<td>3 AUTONOMIC SELF-TESTING</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Test Managers (TMs)</td>
<td>30</td>
</tr>
<tr>
<td>3.1.1 Using MAPE for Testing</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2 Orchestrating Test Interactions</td>
<td>31</td>
</tr>
<tr>
<td>3.1.3 Touchpoint Test Interactions</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4 Implementing Self-Test Controls</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Workflow Integration</td>
<td>40</td>
</tr>
<tr>
<td>3.2.1 Replication with Validation (RV)</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2 Safe Adaptation With Validation (SAV)</td>
<td>43</td>
</tr>
<tr>
<td>4 DESIGN METHODOLOGY TO SUPPORT SELF-TESTING</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Architectural Perspective</td>
<td>46</td>
</tr>
<tr>
<td>4.2 System-Wide Test Coordination</td>
<td>48</td>
</tr>
<tr>
<td>4.3 State-Based Safeness Model</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Formal Definition</td>
<td>51</td>
</tr>
<tr>
<td>4.5 Object Design Goals</td>
<td>53</td>
</tr>
<tr>
<td>4.6 Detailed Object Design</td>
<td>54</td>
</tr>
<tr>
<td>4.6.1 Managed Resources</td>
<td>55</td>
</tr>
<tr>
<td>4.6.2 Autonomic and Test Managers</td>
<td>57</td>
</tr>
<tr>
<td>4.6.3 Management and Testing Interfaces</td>
<td>62</td>
</tr>
</tbody>
</table>
5 CASE STUDY ................................................................. 66
5.1 Applications .......................................................... 66
  5.1.1 Autonomic Container (ACT) ................................. 67
  5.1.2 Autonomic Job Scheduler (AJS) ............................ 69
  5.1.3 Communication Virtual Machine (CVM) ................. 72
  5.1.4 Summary of Applications ...................................... 75
5.2 Experimental Setup ................................................ 76
  5.2.1 Development .................................................... 76
  5.2.2 Performance .................................................... 77
  5.2.3 Test Set Quality ................................................. 78
5.3 Results of Experiments ............................................. 79
  5.3.1 Development .................................................... 80
  5.3.2 Runtime Performance ......................................... 83
  5.3.3 Test Set Quality ................................................. 84
5.4 Empirical Evaluation ............................................... 86
  5.4.1 Observations .................................................... 86
  5.4.2 Lessons Learned ............................................... 88
  5.4.3 Threats to Validity .............................................. 89
6 CONCLUSION ............................................................. 90
  6.1 Research Summary ................................................ 90
  6.2 Contributions ..................................................... 94
  6.3 Future Work ....................................................... 94
APPENDICES ................................................................. 96
BIBLIOGRAPHY ............................................................ 101
VITA .......................................................... 108


LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Self-Management Features of Autonomic Computing</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Comparative Summary of the RV and SAV Validation Strategies</td>
<td>44</td>
</tr>
<tr>
<td>4.1 Design Benefits to Autonomic and Testing Infrastructures</td>
<td>54</td>
</tr>
<tr>
<td>5.1 Descriptive Summary of Applications used in the Case Study</td>
<td>76</td>
</tr>
<tr>
<td>5.2 Mutants Generated to Simulate Faulty Change Requests</td>
<td>80</td>
</tr>
<tr>
<td>5.3 Results of the Mutation Analysis Experiments</td>
<td>85</td>
</tr>
<tr>
<td>5.4 Results of the Program Coverage Analysis Experiments</td>
<td>85</td>
</tr>
<tr>
<td>5.5 Results of the Timed Test Performance Analysis Experiments</td>
<td>85</td>
</tr>
<tr>
<td>6.1 Prototype Coverage of Research Objective 1</td>
<td>91</td>
</tr>
<tr>
<td>6.2 Prototype Coverage of Research Objective 2</td>
<td>92</td>
</tr>
<tr>
<td>6.3 Prototype Coverage of Research Objective 3</td>
<td>93</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Layered Architecture of Autonomic Computing [38]</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Closed Loops of Control in Autonomic Managers, adapted from [38, 76]</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Self-Test Control Loops in Orchestrating TMs</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Two Distinct Self-Test Control Loops in Touchpoint TMs for: (a) Executing test cases on the managed resource, and (b) Performing the post-test evaluation</td>
<td>34</td>
</tr>
<tr>
<td>3.3 High-Level Algorithm of TM Self-Test Controls</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Workflow for Integrating Self-Testing into Autonomic Software</td>
<td>41</td>
</tr>
<tr>
<td>4.1 STAC-Based Perspective of Adaptive Autonomic Software</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Illustrative Example of Self-Test Coordination using STACs: (a) Unit test configuration using stubbed component dependencies, (b) Integration test configuration, and (c) Firewall regression tests for components affected by the change</td>
<td>49</td>
</tr>
<tr>
<td>4.3 State Machine of Safe Adaptation with Validation using STACs</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Formal Definition of a STAC</td>
<td>52</td>
</tr>
<tr>
<td>4.5 Architectural Elements of a STAC</td>
<td>52</td>
</tr>
<tr>
<td>4.6 Class Design for Managed Resources</td>
<td>55</td>
</tr>
<tr>
<td>4.7 Generic Design of Autonomic and Test Managers</td>
<td>58</td>
</tr>
<tr>
<td>4.8 Internal Knowledge Design of Managers</td>
<td>61</td>
</tr>
<tr>
<td>4.9 Autonomic Management Interface</td>
<td>63</td>
</tr>
<tr>
<td>4.10 Automated Testing Interface</td>
<td>64</td>
</tr>
<tr>
<td>5.1 Static View of a Self-Testing Autonomic Container</td>
<td>67</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.2 Self-Test Subsystem of ACT Prototype</td>
<td>68</td>
</tr>
<tr>
<td>5.3 Static View of a Self-Testing Autonomic Job Scheduler</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Activity Diagram of AJS Prototype</td>
<td>71</td>
</tr>
<tr>
<td>5.5 Layered Architecture of the Communication Virtual Machine</td>
<td>72</td>
</tr>
<tr>
<td>5.6 Design of the Self-Configuring NCB Layer of CVM</td>
<td>74</td>
</tr>
<tr>
<td>5.7 Comparison of Infrastructural Development Effort</td>
<td>81</td>
</tr>
<tr>
<td>5.8 Comparison of Incremental ACT and AJS Development Effort</td>
<td>82</td>
</tr>
<tr>
<td>5.9 Comparison of Relative Development Effort for Prototypes</td>
<td>83</td>
</tr>
<tr>
<td>5.10 Percentage of Total Execution Time Taken by AMs and TMs</td>
<td>84</td>
</tr>
<tr>
<td>ACRONYM</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>AC</td>
<td>Autonomic Computing</td>
</tr>
<tr>
<td>ACT</td>
<td>Autonomic Container</td>
</tr>
<tr>
<td>AJS</td>
<td>Autonomic Job Scheduler</td>
</tr>
<tr>
<td>AM</td>
<td>Autonomic Manager</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CUT</td>
<td>Component Under Test</td>
</tr>
<tr>
<td>CVM</td>
<td>Communication Virtual Machine</td>
</tr>
<tr>
<td>DAS</td>
<td>Dynamically Adaptive Software</td>
</tr>
<tr>
<td>MAPE</td>
<td>Monitor, Analyze, Plan, Execute</td>
</tr>
<tr>
<td>NCB</td>
<td>Network Communication Broker</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented</td>
</tr>
<tr>
<td>OAM</td>
<td>Orchestrating Autonomic Manager</td>
</tr>
<tr>
<td>OTM</td>
<td>Orchestrating Test Manager</td>
</tr>
<tr>
<td>RV</td>
<td>Replication with Validation</td>
</tr>
<tr>
<td>SAV</td>
<td>Safe Adaptation with Validation</td>
</tr>
<tr>
<td>SC</td>
<td>Self-Configuration or Self-Configuring</td>
</tr>
<tr>
<td>SH</td>
<td>Self-Healing</td>
</tr>
<tr>
<td>SO</td>
<td>Self-Optimization or Self-Optimizing</td>
</tr>
<tr>
<td>SP</td>
<td>Self-Protection or Self-Protecting</td>
</tr>
<tr>
<td>STAC</td>
<td>Self-Testable Autonomic Component</td>
</tr>
<tr>
<td>TAM</td>
<td>Touchpoint Autonomic Manager</td>
</tr>
<tr>
<td>TTM</td>
<td>Touchpoint Test Manager</td>
</tr>
<tr>
<td>TM</td>
<td>Test Manager</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Continuous technological advances have led to an unprecedented growth in the size and complexity of software systems [39]. The impact of this growth on software maintenance is compounded by the demand for large-scale system integration. Many business operations require collaboration among multiple enterprises, which means that independent software systems will need to be interconnected across administrative domains. Therefore, as technology-driven businesses expand, they incur increasingly high costs to manage and integrate their software solutions [28].

Major industrial players have recognized the need to shift the burden of support tasks such as configuration, maintenance and fault management from people to technology [28, 36, 52]. Microsoft developed the Dynamic Systems Initiative [52]; Hewlett-Packard proposed the Adaptive Infrastructure [36]; and in 2001, IBM introduced the Autonomic Computing (AC) paradigm [43]. A central theme within each initiative is the concept of self-managing software, i.e., computer programs that are capable of governing their own support tasks.

AC emphasizes systems that automatically configure, optimize, heal, and protect themselves [38, 43], in accordance with administrator objectives. IBM has successfully attracted practitioners from the software industry, and members of the academic community, to the field of AC through various manifestos, research papers, and technical reports [28, 38, 39, 42, 43]. In addition, IBM has already incorporated more than four hundred autonomic features into thirty-six distinctive products [39]. However, although AC continues to stimulate great interest in both the industrial and scientific communities, there has been little emphasis on testing autonomic systems.

The next section motivates the need for research projects in the area of testing autonomic systems. More specifically, it emphasizes the importance of software test-
ing during the development of autonomic software, and after deployment. The lack of built-in support for runtime testing in autonomic software is also discussed.

1.1 Motivation

AC poses some grand challenges to the software tester at development time [42, 43]. Incorporating autonomic features into software increases its size and complexity, thereby making it more difficult to validate. Furthermore, each self-management feature must be rigorously tested to ensure that the system: (1) recognizes the conditions in which self-management is required; (2) diagnoses the those conditions correctly; and (3) selects the appropriate course of action to remedy each condition. Pre-deployment testing of autonomic systems may therefore require the formulation of a new testing paradigm, i.e., one that validates the ability of systems to self-diagnose and self-treat [30].

Self-management features may induce structural and behavioral changes to the system at runtime, which has several implications on the reliability of autonomic software. Dynamic software adaptation allows components to be added, removed, or replaced during system operation [79]. If an adaptive change involves integrating a new component, or altering the structure or behavior of existing components, runtime testing should be performed to ensure that: (1) new errors have not been introduced into previously tested components; and (2) the new or adapted component behaves as expected within the context of the software application.

Testing theories and practices provide useful insights on why runtime testing should be incorporated into dynamically adaptive, autonomic software. For example, even when two components are logically equivalent (anti-extensionality), or have the same structural shape (general multiple change), a test set for one is not necessarily adequate for the other [74]. Furthermore, a test set that is adequate for validating a component in isolation, may not be adequate for testing the component’s behavior.
as part of an enclosing component (*anti-composition*) [74]. This is because errors can arise due to interactions between components.

In practice, anti-extensionality and general multiple change are the rationale behind testing software product lines\(^1\), while anti-composition represents the traditional need for integration testing. For autonomic software, each never-before-tested system configuration can be viewed as a member in the same software product line requiring validation. Furthermore, even if the newly introduced or adapted components have been tested separately, failures can occur when the component is placed within the context of the application. Runtime testing should therefore be an integral part of autonomic software to avoid harmful and costly system failures.

Although there is a clear need for runtime testing in autonomic software, there are few research projects on the subject. Most of the active AC research focuses on incorporating the autonomic features of self-configuration, self-optimization, self-protection, and self-healing into domains such as networking, grid computing, and database management. The pioneers of AC state that one of the major challenges associated with building and maintaining autonomic software is validating its correctness [42, 43]. However, the proposed architectural blueprint for autonomic computing provides no support for runtime testing [38].

As enabling technologies of AC continue to advance, it is imperative that researchers exchange ideas on how to validate their dynamic self-management characteristics. This includes: formulating approaches for incorporating runtime testing into autonomic software; studying the detailed designs and prototype implementations that realize these approaches; and sharing the software engineering experiences of conducting such research studies.

\(^1\)Programs that share significant functional and structural commonalities [22]
1.2 Problem Statement

The research problem being explored is in the areas of *software testing* and *autonomic computing*. More precisely, the study focuses on the investigation of an approach that describes the high-level steps for interfacing runtime testing activities with the workflow of autonomic software.

Autonomic software, as specified in the literature [38], lacks the ability to validate structural and behavioral self-management changes at runtime. However, as previously mentioned, several software products have already been developed with autonomic capabilities [2, 5, 34, 39]. A runtime testing approach for AC should therefore be applicable to existing autonomic software systems. Software engineers must be able to quickly identify how, and under what circumstances, the testing approach can be applied to their autonomic software designs and implementations. Furthermore, such designs and implementations may differ in their runtime performance requirements, which calls for flexibility with respect to the operational efficiency of the approach. Lastly, software testing is a complex process that involves many complementary activities. A cross-section of testing activities should therefore be interfaced with autonomic software so the testing effort is realistic.

The research problem is divided into four sub-problems:

1. Formulate an approach that facilitates runtime testing of dynamically adaptive self-configuration (SC), self-optimization (SO), self-protection (SP), and self-healing (SH) changes in autonomic software. The proposed approach should be able to support a subset of key testing activities used in practice.

2. Refine the approach formulated in (1.) by devising strategies that achieve two different levels of transparency during runtime testing — *high transparency*, and

---

2 Timing, memory usage, and CPU utilization
3 Series of maneuvers for obtaining a specific result
low transparency. High transparency means that the effects of runtime testing on the timing and processing characteristics of the software should be negligible. On the other hand, a runtime testing process with low transparency may result in noticeable degradation in the operational efficiency of the software.

3. In cases where the runtime testing process has low transparency, investigate ways in which the negative effects on the processing characteristics of the software can be reduced.

4. Develop a detailed design methodology\textsuperscript{4} to support the implementation of the testing approach formulated in (1.), and the strategies devised in (2.).

An investigation into the aforementioned research problem can lead to: (1) discovery and deeper understanding of the software engineering challenges associated with building reliable autonomic software systems; (2) formulation of algorithms for interfacing runtime testing activities with autonomic software, and minimizing the performance overhead incurred from making these activities available; and (3) creation of reusable autonomic software designs and patterns\textsuperscript{5} for developing self-managing systems that are capable of runtime testing.

1.3 Goal and Objectives

The primary goal of the research presented in this dissertation is to smoothly and seamlessly integrate practical runtime testing activities into autonomic software systems, for the purpose of validating dynamically adaptive self-management changes. This research seeks to address the lack of consideration and support for runtime testing in architectural blueprint for AC [38]. However, since many software products already have autonomic features, testing activities shall be incorporated seamlessly.

\textsuperscript{4}Comprehensive set of design procedures, their usage and underlying rationale

\textsuperscript{5}Known solutions to recurring problems, see Section 2.1.3 for details
to ensure that developers can include them without excessive difficulties. The follow­

ing objectives and evaluation criteria will be used to measure the extent to which the 
research goal has been accomplished:

Objective 1 – Autonomic software designs and implementations, having different per­
formance requirements, shall be able to incorporate test execution, code coverage 
analysis, timed test performance analysis, and test evaluation into their workflow.

Evaluation Criteria – Autonomic designs and prototype implementations extended 
using the approach shall include three or more of the following testing activities:

• Execution of test cases on the component under test (CUT),

• Calculation of line or branch coverage for the CUT,

• Measurement of the time taken to perform test runs,

• Comparison of the results produced by activities 1, 2, and 3 against a predefined 
validation policy.

Objective 2 – Degradation of system performance caused by integrating runtime test­
ing during: (1) high transparency runtime testing was negligible, and (2) low trans­
parency runtime testing was not the result of biased process interleaving in favor of 
testing.

Evaluation Criteria – The criteria for this objective has been divided into two parts 
which correspond to (1.) and (2.) in the previous paragraph.

• Given two autonomic software implementations of the same system, which only 
differ by the following:
:: System 1 $\rightarrow$ \{ No Runtime Testing \}

:: System 2 $\rightarrow$ \{ High Transparency Runtime Testing \}

The difference between the processing and timing characteristics of System 1 and System 2 shall be less than 5%. In addition, there shall be no observable signs of performance degradation due to runtime testing.

- Given an autonomic software implementation in the following category:

:: System $\rightarrow$ \{ Low Transparency Runtime Testing \}

The percentage of the total execution time expended on the test engine shall be less than 50%.

---

Objective 3 – The effort required to integrate runtime testing into an autonomic system, shall be less than the effort required to incorporate its autonomic capabilities.

Evaluation Criteria for Objective 3 – Given an autonomic software system built with runtime testing capabilities: the development effort for supporting and driving the runtime testing activities identified in Objective 1, shall be strictly less than the effort required for building the self-management infrastructure and engine.

The following metrics will be used for comparing the support (infrastructural) aspects of this objective:

- Number of lines of code,

- Number of classes,

- Number of methods,

- Cyclomatic complexity.

In order to compare the relative development effort of the driving (engine) aspects of this objective, the lines of code (LOC) metric will be used.
Summary of Objectives

The aforementioned measurable objectives, in the context of the overall research goal, are summarized as follows:

Objective 1 ascertains that the resulting approach will be applicable to existing autonomic software designs and implementations, and allows it to support a cross-section of testing activities. Objective 2 encompasses optimizations that seek to enhance the runtime efficiency of the approach, thereby promoting seamless operation within autonomic software. Lastly, Objectives 3 is concerned with limiting the software development effort required to build the runtime testing infrastructure and execution engine. This ensures that the implementation of the testing approach is not prominent, thereby allowing the approach to be incorporated smoothly and without inflated software production costs.

1.4 Proposed Solution

To address the research problem defined in Section 1.2, an implicit self-test feature for autonomic software is proposed. The term implicit characterizes the self-testing feature as one that should be inherent in dynamically adaptive autonomic software. The need for self-testing is implied when the autonomic features of SC, SO, SP, and SH can induce structural or behavioral changes to the software. Therefore, the presence of dynamic software adaptation (DSA), or dynamic software updating (DSU\(^6\)), in autonomic software mandates self-testing.

Autonomic self-testing allows the seamless integration of runtime testing into autonomic software. The approach uses the existing AC architecture, as defined in the IBM Architectural Blueprint for AC [38], to conduct runtime testing activities. Using the standard AC architecture for testing has the following advantages: (1) guarantees that the testing approach will be 100% compatible with existing autonomic software,

\(^6\)DSU is a form of adaptation where software updates while it runs [37], e.g., on-line upgrades
as specified by the pioneers of AC; (2) facilitates the reuse of interfaces, source code, policy and log formats, and data repositories; and (3) allows the engineers of autonomic software to use a familiar, uniform, in-progress, software development methods to incorporate runtime testing into their systems. The proposed solution can therefore lead to the rapid realization of a runtime testing framework for autonomic software.

In order to be consistent with the grand vision of AC [43], the proposed self-testing approach automates testing activities wherever possible. However, creating an automated test harness is an extremely challenging endeavor. Software vendors have developed a plethora of tools [13, 17, 20, 26, 32] to aid test automation. Some categories of automated software testing tools are: dynamic analysis tools - apply test cases to the system under test, and test evaluation tools - measure test quality [53]. Under the proposed approach, self-testing is supported by dedicated interface to a collection of dynamic analysis and test evaluation tools.

Self-testing may be incorporated into autonomic software using two general strategies: Replication with Validation (RV) and Safe Adaptation with Validation (SAV). The RV strategy validates adaptive changes using copies of the software components being managed. RV can therefore be implemented on a separate computational node to yield a highly transparent testing process. On the other hand, SAV tests adaptive changes in-place on the actual software components being managed. Autonomic software that employs the SAV strategy may therefore show observable signs of degradation due to the testing process. However, such a strategy may be necessary if it is infeasible to maintain copies of the managed software components.

Finally, the proposed solution is supported by a design methodology, which can be used to implement prototypes of self-testable autonomic software. The design methodology consists of: (1) a logical component-based view of the autonomic system that allows developers to focus on self-management and self-testing concerns; (2) unified modeling language (UML\textsuperscript{7}) schematics, algorithms, and policy formats; and

\textsuperscript{7}UML is a standard graphical notation for specifying abstract models of a software system
(3) detailed guidelines and rationale for applying the design. In this research dissertation, the aforementioned design methodology has been used to realize self-testing in three autonomic software applications.

1.5 Summary of Contributions

This research dissertation establishes the following novel contributions in the area of autonomic computing:


2. Definition of a new, logical, component-based perspective for autonomic software, which allows developers to focus on implementing self-management changes safely and reliably.

3. Provision and analysis of experimental data to support: extending of autonomic software with runtime testing; tailoring runtime testing performance to achieve different levels of transparency in autonomic software; and limiting the development effort required to integrate runtime testing into autonomic software.

1.6 Scope and Limitations

The scope of this research dissertation is confined to the investigation of a testing approach, and supporting design methodology, for enabling runtime testing of adaptive self-management changes in autonomic software. Therefore, the facilitation of runtime testing, as opposed to the attainment of high test adequacy and fault detection levels, is the primary focus of the work. Measurements for test adequacy and fault detection are only provided as a means to demonstrate that runtime testing was indeed performed, and the results evaluated against the testing criteria. Test case
generation to improve the adequacy and effectiveness of testing is outside the scope of this work.

Providing a comprehensive set of runtime testing activities is also beyond the scope of this work. The research goal limits the study to incorporating activities that realize a practical view of the runtime testing effort. For example, software testers recognize that *specification-based evaluation*\(^8\) and *implementation-based evaluation*\(^9\) are equally important. Therefore, facilitating just one category would not reflect a practical testing process. However, realizing a plethora of both types of testing activities is not required to satisfy the research objectives.

While test automation is used to support the proposed solution, the success of the investigation does not depend on the degree of automation. It is widely accepted that many aspects of the software testing process are extremely difficult to automate. The role of automation is simply to provide software engineers with access to testing tools and supporting programs, similar to those used in practice.

Only self-management changes that are dynamically adaptive, and applied to software entities, are considered in the study. Self-management changes that do not modify the structure or behavior of the software at runtime can be validated off-line, prior to system deployment. Lastly, changes to the physical hardware of the computer system are not addressed in this research.

1.7 Outline of the Dissertation

The rest of this dissertation is organized as follows: Chapter 2 provides the background and related work on the problem under investigation. Chapter 3 presents the self-testing approach for autonomic software. Chapter 4 provides the design methodology to support self-testing. Chapter 5 elaborates on the details of the case study. Chapter 6 concludes the research investigation, and discusses future work.

---

\(^8\)Comparing the actual program output with expected output

\(^9\)Measuring how thoroughly the program structure has been exercised
CHAPTER 2

LITERATURE REVIEW

This chapter reviews background information and the current state-of-the-art as it relates to runtime validation of autonomic software. Section 2.1 provides essential material for understanding the problem under investigation; and Section 2.2 compares and contrasts the proposed solution with similar works found in the research literature.

2.1 Background

Recall from Section 1.2 that the research problem is in the areas of autonomic computing and software testing. However, many software design practices have been used to support the proposed solution, and hence are also reviewed in this section.

2.1.1 Autonomic Computing

Autonomic computing (AC) is IBM’s proposed solution to the problems associated with the increasing complexity of computing systems, and the evolving nature of software. The AC initiative was launched in October 2001 and portrayed a vision of computing systems [43] that manage themselves according to high-level objectives. The paradigm seeks to alleviate the burden of integrating and managing highly complex systems through increased automation and goal specification.

The term autonomic is derived from the human autonomic nervous system (ANS), which regulates vital bodily functions without the need for conscious human involvement [39]. For example, when a person enters a hot climate, the ANS automatically induces perspiration to cool the body and maintain a constant temperature (i.e., homeostasis). AC extends the adaptive behavior of the ANS to computing systems and software. Autonomic systems respond to changes in their environment according
to goals set by the system administrator. A dedicated self-management infrastructure is then responsible for maintaining a system state that adheres to those goals. This is achieved by automating low-level decisions and tasks; while allowing administrators to specify system behavior as high-level policies.

The scope of the problem addressed in this research dissertation restricts discussion of AC systems to those in which all managed resources are software entities. Such a system is referred to as an autonomic software system, or autonomic software.

Self-Management Features

The core features that support self-management in autonomic software are provided in Table 2.1 [43, 54]:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Configuration</td>
<td>SC</td>
<td>Automatically configuring or re-configuring existing system components, and seamlessly integrating new components.</td>
</tr>
<tr>
<td>Self-Optimization</td>
<td>SO</td>
<td>Automatically tuning resources and balancing workloads to improve operational efficiency.</td>
</tr>
<tr>
<td>Self-Protection</td>
<td>SP</td>
<td>Proactively safeguarding the system against malicious attacks, and preventing damage from uncorrected cascading failures.</td>
</tr>
<tr>
<td>Self-Healing</td>
<td>SH</td>
<td>Proactively discovering, diagnosing, and repairing problems resulting from failures in hardware or software.</td>
</tr>
</tbody>
</table>

Table 2.1: Self-Management Features of Autonomic Computing

Architecture

The architectural blueprint for AC [38] defines a common layered approach for developing self-managing systems as shown in Figure 2.1. The horizontal layers (from bottom to top) include: managed resources, touchpoints, touchpoint autonomic managers, orchestrating autonomic managers, and a manual manager. A vertical layer of knowledge sources (top-left of Figure 2.1) spans the top three horizontal layers to facilitate the exchange and archival of management information.
The managed resource layer consists of the software entities for which self-management services are being provided. Directly above the managed resources are manageability interfaces called touchpoints. Touchpoints implement the sensor and effector behaviors necessary to automate low-level management tasks [38, 43]. Sensors provide mechanisms for observing the state of managed resources, while effectors facilitate the implementation of runtime changes.

A higher level of management is provided by autonomic managers (AMs). There are two categories of AMs – Touchpoint AMs, and Orchestrating AMs [38]. Touchpoint AMs work directly with managed resources through their touchpoints. Orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for individual resources. Orchestration may therefore occur within a single discipline for multiple resources (e.g. SC only), or across multiple disciplines for a single resource.

The topmost layer is an implementation of a management console, called the manual manager, which facilitates the human administrator activity. Lastly, the vertical layer of knowledge sources implements registries or repositories that may be used to extend the capabilities of AMs, and are directly accessible by the human administrator via the manual manager layer.
Autonomic Control Loops

Autonomic software systems are characterized by closed loops of control. Sensed changes to managed resources result in the invocation of a set of actions designed to maintain some desired state. Autonomic control loops are implemented as monitor, analyze, plan, and execute (MAPE) functions in AMs [38].

As shown at the bottom of Figure 2.2, different kinds of software users can affect the state of managed resources. These users include human and non-human (e.g., file system, operating system, other software components) external entities that submit inputs to the managed resource [76]. The MAPE functions of AMs collaborate to manage state changes to the resource as follows:

- Monitor – continuously polls the managed resource for this state information, and correlates it into symptoms for analysis.

![Autonomic Manager Diagram](image-url)
- Analyze – determines if the current state is undesirable, and generates a change request to be passed to the plan function.

- Plan – specifies the set of actions needed to remedy the state condition of the managed resource, and formalizes them into a plan for execution.

- Execute – implements change plans on the managed resource through its effectors, for the purpose of acquiring some desired state.

- Knowledge – coordinates access to data shared among the MAPE functions.

High-level coordination of the MAPE functions is achieved through a hierarchical juxtapositioning of AMs. As shown at the top of Figure 2.2, the state of the MAPE functions and internal knowledge may also be observed and manipulated through sensors and effectors. Orchestrating AMs can therefore detect the generation of MAPE artifacts, and determine alternative courses of action. In addition, the self-management policies that guide the behavior of AMs may be dynamically updated through these top sensors and effectors. In summary, the design of autonomic software is highly flexible and extensible through a hierarchy of automated management activities.

Dynamic Software Adaptation

Autonomic changes may involve structural and behavioral modifications to the software at runtime, referred to as dynamic software adaptation (DSA). DSA requires that software systems be capable of observing their own structure and behavior (introspection); modifying their own structure and behavior (intercession); and identifying their specific environmental and operational conditions (context-awareness) [79].

The construction of dynamically adaptive software can be realized through a variety of development practices. Software architectures such as Reflection [15] facilitate the modification of fundamental aspects of the software, including type structures
and function calls. Many programming languages provide implementation-specific support for introspection and intercession through application programming interfaces (APIs). For example, at runtime the Java Reflection API [66] can be invoked to identify the class types Java objects; and determine their constructors, fields, modifiers, and superclasses. Component-based frameworks such as (e.g., Spring [73]) can also be combined with scripting languages (e.g., Groovy [46]) to enable DSA.

Ensuring Safeness

One of the major challenges associated with DSA is ensuring the safety and integrity of the adaptation process. Zhang et al. [79] proposed a disciplined approach to DSA, referred to as safe adaptation. Safe adaptation guarantees that the adaptation process does not violate any dependency relationships among components, or interrupt critical communication segments.

The safe adaptation process is comprised of three phases: analysis, detection and setup, and realization. During analysis, developers prepare a data structure for holding information such as component configurations, dependency relationships, and adaptive actions. The detection and setup phase occurs at runtime and involves generating safe adaptation paths for performing adaptive actions on system components. Component adaptation is then performed as part of the realization phase. The major steps of the realization phase are as follows:

1. Move the system into a partial operation mode in which some functionalities of the component(s) to be adapted are disabled.

2. Hold the system in a safe state while adaptive actions are performed.

3. Resume the system's partial operation once all adaptive actions are complete.

4. Perform a local-post action to return the system to a fully-operational state.
If a failure occurs at any point before Step 3, the adaptation manager can retry the failed actions or **rollback** to the source configuration. Rollbacks are not allowed after any system process has been resumed. Therefore, safe adaptation is atomic in the sense that either no side effects are produced or the adaptation process runs to completion.

### 2.1.2 Software Testing

Software testing is the process of operating software under specified conditions, observing or recording the results, and making an evaluation of some aspect of the software [40]. Testing involves executing a finite set of test cases on the software system, and comparing the results against predefined criteria to determine whether tests passed or failed. This section overviews some main categories of software testing, and describes the notion of test adequacy criteria.

**Black Box vs. White Box Testing**

The distinction between these two categories is based on whether or not the tester has knowledge of the internal workings of the software. In **black box** or **specification-based** testing, the tester employs no knowledge of the program structure when developing test cases. Testing is therefore based solely on the program specification [6]. On the other hand, **white box** or **program-based** testing focuses on whether or not elements of the program structure (e.g., lines, branches) have been thoroughly exercised [6].

**Unit, Integration, and System Testing**

Testing can occur at the **unit**, **integration**, and **system** levels [6]. Unit testing validates individual software components in isolation; while integration testing checks for faults that can arise from interactions between multiple components. During integration, the testing team may be required to: (1) develop test stubs, i.e., mock implementations of components for the purpose of testing, and (2) generate an integration
test order (ITO) to minimize the number of stubs needed [19]. Once integration is complete, a series of system tests are executed to validate the system as a whole.

Regression Testing

The continual evolution of software means that systems typically require perfective, adaptive, or corrective maintenance after initial deployment [62]. Regression testing determines whether modifications to software have introduced new errors into previously tested code [6]. This may involve re-running the entire test suite (retest-all), or a strict subset (selective retest) [35]. A selective retest method commonly used in practice is firewall regression testing [75]. Firewall regression testing uses change impact analysis to identify the set of components affected by the change [75]. The identified components are then retested to ensure the system still behaves as intended.

Manual vs. Automated Testing

Software testers can manually apply test inputs to the software in a similar fashion to end-users. However, several tools have been developed to aid the automation of the testing process [13, 17, 26, 32]. Test automation involves creating test scripts; and setting up a test harness for executing tests, logging the results, and performing a post-test evaluation [53]. If the post-test evaluation passes then the test harness should automatically terminate, otherwise additional test cases should be selected and fed through the harness with the aim of improving the testing effort.

Mutation Testing

The effectiveness of a test set with respect to its ability to reveal faults can be assessed using a technique known as mutation testing [6, 82]. Mutation testing involves generating a set of programs or specifications, called mutants, that differ from the original program or specification in some way. The test set is then executed using the
mutants and the results are compared with those produced from using the original program or specification.

For each mutant, if the test results differ from the original results on at least one test case, the mutant is said to have been killed; otherwise the mutant is still alive [82]. A mutation score can then be used to measure test adequacy by calculating the ratio of the number of dead mutants over the total number of mutants that are not equivalent to the original program or specification.

Test Adequacy

The notion of test data adequacy is central to any testing approach because it can be viewed as a measurement of test quality, and as a guideline for the generation of test cases. If $P$ is a set of programs, and $S$ is a set of specifications, and $T$ is a set of test cases, we can formally define a test data adequacy criterion $C$ as follows [82]:

- **Measurements** - $C : P \times S \times T \rightarrow [0, 1]$. $C(p, s, t)$ maps to a real number representing the degree of test adequacy; and the greater the value, the more adequate the testing.

- **Generators** - $C : P \times S \rightarrow 2^T$. $C(p, s)$ maps to a power set containing all test cases that satisfy the criterion, and hence any element is adequate for testing.

Test data adequacy criteria can also be viewed as a stopping rule that determines whether or not enough testing has been done [82]. However, as a stopping rule, a test data adequacy criterion $C$ is a special case of measurements with the range $\{0, 1\}$.

2.1.3 Software Design

The software design process involves defining the architecture, components, interfaces, and other fundamental characteristics of a software system [40]. During software design, developers formulate and specify a software solution to a problem.
Objects and Components

Several abstraction techniques have been developed for specifying software systems. The object-oriented (OO) [14] paradigm is currently the de-facto standard used in industry. In OO software, the system is viewed as a set of objects that collaborate through message passing. Each object has:

- A Reference – a unique identifier for the object;
- Attributes – a set of data types that represent the object’s characteristics;
- Methods – a set of operations that define the object’s behavior;
- State – a mapping that associates each attribute with a current value.

The component perspective offers a higher-level of abstraction than objects [14]. Components encapsulate multiple objects into a single element that provides multi-use, non-context-specific services at the system level. Developers can therefore purchase commercial-off-the-shelf (COTS) components, and tailor the services provided to the specific needs of the application.

Reuse and Patterns

The main forms of software reuse include: interface reuse – reusing the signatures available for message passing; code reuse – reusing classes or collections of procedures and functions; and pattern reuse – reusing solutions to well-known problems [33].

Reuse is a central theme in object-oriented software engineering (OOSE), and component-based software engineering (CBSE). The concepts of composition and delegation are common to both OOSE and CBSE. Composition provides a way to combine objects, components, or services into more complex ones; while delegation dynamically binds a method call for an object to another object.\(^\text{10}\)

\(^{10}\)Or in CBSE, a service call for a component to another component
The notion of inheritance can be used to reuse class interfaces and implementations in OO software. Given two classes $A$ and $B$ in an OO system, $B$ derives $A$ means that the attributes and behaviors of $A$ will be present in $B$. Similarly, if $A$ and $B$ are interfaces then the method signatures declared in $A$ will be available in $B$. Inheritance is highly flexible because the implementation of members in one class, referred to as an abstract class, can be deferred to one or more of its concrete derived classes [19].

Software patterns allow the reuse of knowledge gained from past development activities. They provide detailed descriptions that capture the experience of software engineers, when designing or implementing a solution to a recurring problem. The following types of patterns are commonly cited in the literature [15, 33]:

- **Architectural Patterns** – express fundamental structure of software using a set of predefined subsystems and their responsibilities;

- **Design Patterns** – refines the subsystems or components of the software and the relationships between them;

- **Idioms** – describes the low-level implementation of components using the features of a given programming language.

### 2.2 Related Work

Runtime testing as an integral feature of self-managing software has received little attention in the research community. Most of the works that target the AC paradigm focus on the core self-management properties of self-configuration, self-protection, self-optimization, and self-healing. However, many aspects related to the problem under investigation and proposed solution have been studied in the literature.

---

11Concrete classes are classes that have no deferred members
2.2.1 Self-Testable Software

Several researchers have investigated the notion of self-testing software [12, 23, 24, 70]. Blum et al. [12] introduced a technique which uses self-testing/correcting pairs to verify a variety of numerical problems. The idea is that a user can take any program and its self-testing/correcting pair of programs and, once the program passes the self-test, on any input call the self-correcting program which will make the appropriate calls to the original program to compute the correct value. If this notion of self-testing/correcting program pairs worked for complex programs then self-testing for autonomic computing systems would be trivial. However, this technique only works for very well defined functions.

Denaro et al. [24] present an approach that automatically synthesizes assertions from the observed behavior of an application with the objective of adaptive application monitoring. The proposed approach embeds assertions into the communication infrastructure of an application that describes the legal interactions between the communicating entities. These assertions are then checked at runtime to reveal misbehaviors, incompatibilities, and unexpected interactions that may occur because of hidden faults. The focus of the work is to provide systems with the ability to automatically synthesize assertions that evolve over time and adapt to context-dependent interactions. The synthesis of assertions at runtime can benefit the self-testing of adaptive systems by providing a way to generate additional test cases for validating components after an autonomic change has been implemented.

The work by Le Traon et al. [70] is closely related to our work; it describes a pragmatic approach to develop self-testable components that link design to the testing of classes. Components are self-testable by including test sequences and test oracles in their implementation. For this approach to be practical at the system level, structural test dependencies between self-testable components must be considered at the system architecture level. Our work extends the work by Le Troan et al. [70] by
considering how the architectural perspective of self-testable components can be used to realize unit, integration, system, and firewall test execution in autonomic software.

2.2.2 Testing Adaptive Systems

Software testing continues to be the primary means of validation\textsuperscript{12} used in industry, and hence there has been much research on validating software adaptation and variation. However, very few works address specific self-managing paradigms such as autonomic computing. A comparative summary of the works related to testing dynamically adaptive software, and families of software products is therefore presented.

Environmental Variations

Munoz and Baudry [55] presents an approach to testing the adaptation logic called Artificial Shaking Table Testing (ASTT). Using ASTT the authors are able to select test data that simulates complex environmental variations called context-shakes that can reveal faults in the adaptation policy. The synthesis of context-shakes is modeled as a search problem, and a series of shakes are automatically generated for finding faults associated with environmental completeness, and adaptation correctness. ASTT is applied to an adaptive web server application, and mutation analysis is used to simulate errors in the software adaptation policies for experimental evaluation.

The modeling and synthesis techniques provided by ASTT [55] are highly complementary to the research problem addressed in this dissertation. From a broad perspective, validating the correctness of dynamically adaptive software is bipartite and may therefore be separated into two distinct research topics: (1) determining whether or not the software is adapting correctly to environmental changes; and (2) ensuring that the software behaves correctly after an adaptive change has been applied. The former is addressed by approaches such as ASTT, while the latter is facilitated through research investigations such as the one in this dissertation.

\textsuperscript{12}Validation determines if the software meets the needs of the customer
Both ASTT and self-testing in autonomic software can be deemed as equally important channels of research investigation. The rationale for the previous statement is analogous to the need for specification-based and implementation-based software. For example, even if ASTT determines that an adaptation is correct with respect to environmental conditions, the target configuration may have introduced errors into the software system. Conversely, new configurations that have passed runtime self-tests may have been incorrect with respect to environmental changes.

Software Variations

Developers of adaptive systems must specify the parts of the system that will be allowed to vary at runtime. An area of software research and practice that addresses the specification of software variation is Software Product Lines (SPL) [21]. SPL proposes uses variation points to determine the changeable structure of the software while maintaining its overall architecture. Adaptation is therefore driven by logics that compute the correct variant that should be adopted in given environmental conditions.

Several researchers have been investigating testing approaches for validating SPLs [22, 50, 57, 67]. Some works focus on harnessing the unique features of the software architectural abstractions for a variety of test objectives. This includes research on: detecting mismatches in components [3]; developing integration test plans [8]; and formulating test coverage criteria based on the architectural model [59]. There is a significant body of work on limiting the number of tests that need to be performed for the members of the same SPL [22, 50]. The major concerns addressed with such approaches are the increase in workload for the programmer, and the length of time spent to perform the tests.

Research on testing SPLs is closely related to the work presented in this dissertation. However, the notion of self-testing in autonomic software carries many of the aforementioned testing concerns into the runtime environment. Self-testing in auto-
onomic software provides embedded mechanisms for incorporating the fault detection, and test optimization techniques for SPLs in the system’s runtime behavior.

Merging the two approaches can lead to several optimizations in building and operating self-testable autonomic software. Research on testing SPLs can be used to further reduce the development and performance overhead of autonomic self-testing. In particular, if testing must be performed in-place during system execution, then reducing the test set is of utmost importance for maintaining quality of service. Lastly, the emerging research area of Dynamic Software Product Lines (DSPLs) has clear overlaps with self-managing software, and will therefore benefit from the research investigations on autonomic self-testing.

2.2.3 Model Checking Adaptive Systems

Verification is the process of determining whether or not a software product satisfies its specification [18]. Large emphasis in the research community is placed a formal verification technique known as model checking. The model checking problem can be defined as follows [18]: Given a model $M$ and a temporal logic formula $\varphi$, find the set of states $s \in S \mid M, s \models \varphi$. In the model checking definition, the symbol $\models$ represents a “satisfies” relationship.

Model checking has the advantage that it can be automated, and therefore used at runtime to generate witnesses or counterexamples\textsuperscript{13}. On the other hand, a disadvantage of model checking is that it is limited to reactive, finite state\textsuperscript{14} systems. However, model checking has proven to be a viable approach for improving the reliability of adaptive software.

Zhang et al. [80] tackle the problem of verifying adaptive systems using modular model checking. They attempt to address the state space explosion problem by confining

\textsuperscript{13}Execution paths where a property of the software is either satisfied, or violated

\textsuperscript{14}Both model checking, and testing suffer from the state space explosion problem [71]
model checking activities when possible to only those aspects of the system affected by the change. Model checking is performed using a finite state machine to check transitions between software variations. The authors have also used Petri Nets to verify system properties and behaviors for dynamically adaptive systems [78].

The work by Zhao et. al [81] present a model-based runtime verification technique that can be applied to self-optimizing systems. More specifically, their approach targets component-based, self-optimizing systems. At runtime, self-optimization is performed in a real-time operating system by dynamically exchanging components. Model checking is performed to check time-annotated, temporal logic properties on real-time (RT) UML models. The on-line model checking, which operates as a service of the underlying OS, is interleaved with the components to determine if substitutions are safe and consistent.

Kulkarni and Biyani [47] developed a formal approach that uses a proof-lattice to verify that all possible adaptation paths do not violate global constraints. Their later work on testing distributed systems [11] applies predicate detection techniques for testing adaptive systems at runtime. Existing algorithms for global predicate evaluation are extended for checking the correctness of distributed systems during dynamic adaptation.

Each of the aforementioned approaches share commonalities with the research problem and solution presented in this dissertation. In particular, the works by Zhang et. al [78, 80] and Zhao et. al [81] are most closely related to testing dynamically adaptive, autonomic software. One of the validation strategies proposed in this dissertation builds on the early works of Zhang, which facilitates safe component-based adaptation [79]. However, as opposed to extending such work for the purpose of model-checking [80], autonomic self-testing uses safe adaptation to provide local and global control mechanisms for runtime validation.

The approach presented in Zhao et. al [81] is also very similar to the notion of autonomic self-testing, but focuses on model checking self-optimizing software. How-
ever, autonomic self-testing is presented as a general approach that is also applicable self-configuration, self-healing and self-protection features. A disadvantage of performing testing in-place is that some functionalities of the software may have to be blocked until validation completes. Furthermore, testing operations tend to be more computationally expensive than those involving abstract models. Therefore, the runtime overhead of autonomic self-testing can be reduced by combining it with the model checking approach of Zhao et. al [81], which already targets self-optimizing software systems.
CHAPTER 3

AUTONOMIC SELF-TESTING

This chapter presents a novel approach for seamlessly integrating runtime testing activities into autonomic software. The approach introduces an implicit self-test feature into autonomic software to validate dynamically adaptive, self-management changes.

Self-testing is incorporated into autonomic software by tailoring *autonomic managers* and *knowledge sources* to runtime testing activities. Autonomic managers (AMs) are specialized for testing by implementing control loops that monitor the state traditional AMs. These specialized AMs, referred to as *test managers* (TMs), are designed to intercept the adaptive change requests generated in AMs.

TMs perform runtime testing on adaptive self-management changes before they are implemented on managed resources. They are responsible for: (1) validating the behavior of newly added or adapted software components prior to their use in the system; and (2) performing regression testing to ensure that new errors have not been introduced into previously tested components. In addition, the self-testing behavior of TMs is supported through an interface for setting up and invoking services of automated software testing tools.

Knowledge sources are also specialized for testing purposes. Test artifacts such as validation policies, test logs, and test histories are stored and organized in *test knowledge sources*. The behavior of TMs is therefore extensible through the use of test knowledge sources. For example, updated validation policies can be transferred to TMs at runtime to extend their capabilities.

This chapter is organized as follows: Section 3.1 describes the duties and interactions of test managers (TMs), and provides two detailed algorithms for using TMs to realize autonomic self-testing control loops. Section 3.2 presents a step-by-step workflow for integrating TMs into autonomic software through interactions with
AMs. The workflow is presented in the context of two strategies: Replication with Validation (RV), which provides a highly transparent self-testing process; and Safe Adaptation with Validation (SAV), which is less transparent but offers an alternative solution under special circumstances.

3.1 Test Managers (TMs)

Similar to autonomic managers, TMs may be Orchestrating or Touchpoint [38]. Orchestrating TMs direct high-level testing activities, and manage Touchpoint TMs. On the other hand, Touchpoint TMs perform low-level testing tasks on managed resources. During system execution, TMs monitor AMs to determine when autonomic change requests require validation. TMs may be used to perform the following:

- Test Coordination – Directing runtime validation through the orchestration of low-level testing tasks. TMs set up and coordinate the following entities: AMs, Touchpoint TMs, test knowledge sources, and automated testing tools.

- Test Planning and Execution – Developing and implementing runtime validation plans for managed resources. This includes selecting test cases, and scheduling the performance of tests.

- Test Suite Management\(^{15}\) – Maintaining up-to-date test sets and test scripts. TMs are responsible for dynamically generating test cases for managed resources, and discarding existing tests that are no longer applicable due to structural or behavioral changes.

- Pre- and Post-Test Setup – Setting up the test environment, and preparing a log of the test results for the post-test evaluation.

\(^{15}\)Outside research scope but included here for completeness of the testing approach.
• Post-Test Evaluation – Analyzing and evaluating test results including number of test passes/failures, and coverage of the implementation; against a predefined validation policy.

• Storage of Test Artifacts – Maintaining a repository to store test cases, test logs, test histories, and validation policies.

3.1.1 Using MAPE for Testing

To perform the aforementioned activities, TMs apply the MAPE functions [38] of AC to runtime testing as follows:

A test monitor function is responsible for polling the different entities within the autonomic software system and collects information relevant to the validation process. A test analyzer function determines whether or not some testing-related activity needs to be performed, such as setting up the test environment, selecting test cases, or conducting the post-test evaluation. A test planner function then develops a plan for realizing the testing activity, and a test executer function carries out the specified tasks. A test knowledge component serves as a central repository for test artifacts, and coordinates the interactions between the MAPE functions during runtime testing.

3.1.2 Orchestrating Test Interactions

Figure 3.1 illustrates how the autonomic control loops of Orchestrating TMs implement high-level test coordination. As previously mentioned, test coordination involves directing complex interactions between multiple entities within self-testable autonomic software; as shown at the bottom of Figure 3.1. These entities include Orchestrating AMs, Test Knowledge Sources, Touchpoint TMs, and the Automated Testing Tool Support. Information gathered from these entities is passed through the MAPE functions of the Orchestrating TM to accomplish specific testing objectives;
Figure 3.1: Self-Test Control Loops in Orchestrating TMs.

as indicated by the lines labeled OTM that enter and exit through the sensors and effectors in Figure 3.1. The workflow of Figure 3.1 is as follows:

- Orchestrating TMs continuously poll Orchestrating AMs (OTM 1a) to detect when the autonomic software requires validation. Test knowledge sources may also be monitored (OTM 1b) to detect when validation policies are updated.

- These events can be handled through a control loop that uploads the appropriate validation policy a Touchpoint TM (OTM 2), thereby initializing the new or requested testing functionality.

- After initialization, Touchpoint TMs are monitored to detect when they require support tools such as code coverage profilers to be configured (OTM 3) for testing.

- Configuration of the support tool(s) is realized by invoking the interface of the automated testing tool support (OTM 4).

- Once the tool has been successfully configured, the Orchestrating TM detects this event (OTM 5) and sends a notification to the Touchpoint TM (OTM 6).
Other possible orchestrating interactions\(^\text{16}\) include: (1) receiving notifications from Touchpoint TMs as to whether validation passed or failed (OTM 7a), or if there was inadequate test coverage (OTM 7b); and (2) responding to those notifications by either terminating the testing process (OTM 8a), or continuing testing (OTM 8b) in an attempt to improve test coverage.

### 3.1.3 Touchpoint Test Interactions

Figures 3.2(a) and 3.2(b) show the behavior of two self-testing control loops within Touchpoint TMs. The first loop traces the arc labeled with artifacts 1.1 through 1.5 in Figure 3.2(a), and the second loop traces the arc labeled with artifacts 2.1 through 2.5 in Figure 3.2(b). The operations of Touchpoint TMs can be coordinated by an Orchestrating TM; as indicated by the lines labeled with the prefix OTM which enter and exit through the top sensors and effectors, respectively.

The workflow and relationship between Figure 3.1, and the first self-testing control loop shown in 3.2(a) are described as follows:

- The Orchestrating TM uploads a validation policy (OTM 2) into the internal knowledge of the Touchpoint TM (1.0). This policy contains details on the managed resource to be tested, its adaptive change request, and testing criteria.

- The Touchpoint TM automatically invokes its monitor function to gather state information relating to whether or not the managed resource has been set up for validation (1.1).

- When the analyze function (1.2) determines that the test setup actions for the managed resource are complete, a test suite is prepared for validating the structural or behavioral change using: data on the change request; knowledge of the previous structure or behavior of the managed resource; and a baseline test suite.

\(^{16}\) Not shown in Figure 3.1 but the labels are referenced in Figures 3.2 and 3.3
Figure 3.2: Two Distinct Self-Test Control Loops in Touchpoint TMs for: (a) Executing test cases on the managed resource, and (b) Performing the post-test evaluation

- The analyze function then notifies the test planner that the test suite is ready, and requests that a test plan be created (1.3).

- The Orchestrating TM detects that a new test plan (1.3.1) has been generated, and suspends the control loop to set up any automated tools to support testing (OTM 3).

- When the support tools have been successfully configured for use with the managed resource (OTM 6), the self-test control loop resumes and the test plan is finalized (1.3.2).

- The test plan is passed to the execute function (1.4), which then invokes the test interface of the managed resource to run the tests (1.5).

Similarly, the workflow and relationship between Figure 3.1, and the second self-testing control loop shown in 3.2(b) are described as follows:
After test execution, the monitor function coalesces the test results for the managed resource (2.1) into a log file, which is accessed via the test interface of the managed resource.

The test log is then passed to the analyze function for the post-test evaluation (2.2).

During the post-test evaluation, the analyze function compares the test results against the testing criteria stored in the validation policy. The goal is to determine whether or not validation passed or failed with respect to: functional outputs (2.2.1a, OTM 7a); and coverage of the program structure (2.2.1b, OTM 7b).

Orchestrating TM makes the final decision regarding whether or not to end the validation process (2.2.2a), or continue testing in an attempt to achieve the desired level of coverage (2.2.2b).

If a decision is made to continue testing, the behavior of the second loop, from 2.2 through 2.5 in Figure 3.2(b), is the same as the first loop from 1.2 to 1.5 in Figure 3.2(a).

Figure 3.3 combines the self-test actions and interactions of Orchestrating TMs (Figure 3.1) and Touchpoint TMs (Figure 3.2) into a high-level algorithm. The algorithm assumes that Orchestrating TMs can deploy Touchpoint TMs on-demand, i.e., upon request or as needed. However, while the algorithm in Figure 3.3 represents an end-to-end view of the testing process, more details are required to implement the behavior of both types of TMs. It is therefore necessary to refine this high-level algorithm into detailed algorithms for Orchestrating TMs and Touchpoint TMs.
3.1.4 Implementing Self-Test Controls

Algorithms 3.1 and 3.2 provide the detailed steps for implementing self-test controls in Orchestrating TMs and Touchpoint TMs, respectively.

Orchestrating Self-Test Control

Input to Algorithm: (1) oam – object reference to the Orchestrating AM that generates adaptive change requests; (2) ttm – object reference to an available Touchpoint TM; and (3) policy – validation policy for guiding the decisions of the Orchestrating TM that implements this algorithm.

Algorithm 3.1: Orchestrating Self-Test Control Algorithm

```
1: OTM_Self_Test_Control(oam, ttm, policy)
2: loop
3: symptom ← monitor(oam, ttm)
4: if (“OAM Change Generated” ∈ analyze(symptom, policy)) then
```
oamChange ← oam.requests.dequeue()
if (oamChange ∈ “Adaptive”) then
    plan.enqueue(“Validation Required”, oamChange)
end if
end if
if ("TTM Result Generated" ∈ analyze(symptom, policy)) then
    ttmResult ← execute(ttm.requests.dequeue())
    if ("Functional Fail" ∈ ttmResult) then
        plans.enqueue("OAM Reject Notice", ttmResult)
    else
        if ("Functional Pass" ∈ ttmResult) then
            if ("Adequate Coverage" ∈ ttmResult) then
                plans.enqueue("OAM Accept Notice", ttmResult)
            else
                otmResult ← analyze(ttmResult, policy)
                if ("Coverage Fail" ∈ otmResult) then
                    plans.enqueue("OAM Reject Notice", otmResult)
                else
                    plans.enqueue("TTM Continue Notice", otmResult)
                end if
            end if
        end if
    end if
end if
if (plans ≠ ∅) then
    nextPlan ← plans.dequeue()
    if ("Validation Required" ∈ nextPlan) then
        execute(oam.setupTest(oamChange))
        baseTests ← lookupTests(oam.change.resource))
        ttmPolicy ← lookupPolicy(oam.change.resource))
        execute(ttm.init(oamChange, baseTests, ttmPolicy))
    end if
    if ("OAM Reject Notice" ∈ nextPlan) then
        execute(oam.setAccept(false))
    end if
    if ("OAM Accept Notice" ∈ nextPlan) then
        execute(oam.setAccept(true))
    end if
    if ("TTM Continue Testing" ∈ nextPlan) then
        attentionFlag ← true)
    end if
end if
end if
end loop
Continuous autonomic self-test control for the Orchestrating TM (OTM) is represented by the loop spanning lines 2 through 47. In Line 3, the function call \texttt{monitor(oam, ttm)} commences the OTM's polling activities, which involves the retrieval of symptom data from both the Orchestrating AM and the Touchpoint TM. Symptom detection is specified from lines 4 through 28, which compare the retrieved symptom data against known symptoms defined in the OTM's policy via the function call \texttt{analyze(symptom, policy)}. Two types of symptoms can be detected by the algorithm: (1) OAM generation of a change request, specified in lines 4 through 9; and (2) TTM generation of test results from a completed test run, described by lines 10 through 28.

In the case of OAM change request generation, if the change involves dynamic adaptation (line 6) then the request is sent to a queue for the plan function indicating that validation is required. However, TTM result generation involves three courses of action according to whether the functional and code coverage test requirements have been met (lines 12, 15 and 16), or if the test failed due to inadequate code coverage (line 20). Each of these cases is immediately followed by the enqueuing of its respective change plan for accepting, rejecting or continuing the testing process.

The handling of queued change plans is described by lines 29 through 46 of the algorithm. Plans for validation result in setting up the pretest state of the managed resource (line 32); selecting test cases and searching for a validation policy for the resource (lines 33 and 34), and deploying the TTM to perform self-testing (line 35). For the remainder of the algorithm, the OTM provides feedback to the OAM as to the results of testing. A special point to note is that when there is inadequate coverage, the algorithm simply sets an \texttt{attentionFlag} in line 44 to alert the administrator of this event. This is because dynamic test planning and test case generation are outside the research scope.
Touchpoint Self-Test Control

Input to Algorithm: (1) resource -- object reference to the managed resource to be tested, (2) changeData -- information relating to the adaptive change request (3) baseTests -- a set of test cases for the managed resource; and (4) policy -- rules for guiding the testing behavior of the Touchpoint TM that implements this algorithm.

Algorithm 3.2: Touchpoint Self-Test Control Algorithm

1: TTM_Self_Test_Control(changeData, baseTests, policy, resource)
2: loop
3: inSymptom ← monitor(resource)
4: if (“Ready for Test” ∈ analyze(inSymptom, policy)) then
5:   requests.enqueue(“Test Execution”, inSymptom)
6: end if
7: if (“Testing Complete” ∈ analyze(inSymptom, policy)) then
8:   testResult ← analyze(testLog, policy)
9:   requests.enqueue(“OTM Result Notice”, testResult)
10: end if
11: if (requests ≠ ∅) then
12:   nextRequest ← requests.dequeue()
13:   if (“Tests Execution” ∈ nextRequest”) then
14:     testSet ← plan(changeData, baseTests)
15:     plans.enqueue(“Test Execution”, testSet)
16: end if
17: end if
18: if (plans ≠ ∅) then
19:   nextPlan ← plans.dequeue()
20: if (“Test Execution” ∈ nextPlan”) then
21:   testLog ← execute(resource.test(testSet))
22: end if
23: end if
24: end loop

Continuous autonomic self-test control for the Touchpoint TM (TTM) is represented by the loop spanning lines 2 through 24. Polling of the managed resource is defined in Line 3 by the function call monitor(resource). Two of the following symptoms for the resource can be recognized: Ready for Test – pretest state for the resource has been set and therefore testing can begin (lines 4–6), and Testing
Complete – test execution has completed and the results of testing are ready to be evaluated (lines 7 to 9).

The block from line 11 to 17 checks the queue of requests for validation, and generates a test plan. A test set is selected through a call to the function `plan(changeData, baseTests)` (line 14), and then queued for execution (line 15). Lastly, the generated test plans are dequeued and the function call `execute(resource.test(testSet))` applies the test cases to the managed resource (line 21).

3.2 Workflow Integration

Figure 3.4 provides a detailed workflow for integrating self-testing into autonomic software. The autonomic software system is shown on the left of Figure 3.4, while the self-testing infrastructure is on the right. Self-testing may be incorporated into autonomic software using two strategies: Replication with Validation, (RV) and Safe Adaptation with Validation (SAV).

RV and SAV may be employed in the same autonomic software system. However, they are designed to be mutually exclusive per managed resource, i.e., a managed resource can be associated with exactly one strategy at any instance in time. Furthermore, the two strategies differ with respect to their: development and maintenance costs; hardware requirements; assumptions and applicability under special circumstances; and level of transparency to the autonomic software system.

3.2.1 Replication with Validation (RV)

The RV strategy requires access to copies of managed resources, solely for the purpose of runtime testing (dashed box at bottom-left of Figure 3.4). Therefore, developers of autonomic software must either deploy the copies with the system, or build the system with mechanisms for replicating its resources at runtime. In addition, RV
assumes that the autonomic system will be able to apply self-management changes to these copies, as a preliminary step in setting up the testing environment.

The workflow for integrating self-testing into autonomic software using RV is as follows (starting from the bottom-left of Figure 3.4):

1. **Ready for Self-Management** – A Touchpoint AM gathers state information from a managed resource and correlates it into symptoms that warrant self-management.

2. **Change Request Detected** – A change request is generated by the Touchpoint AM and this event is detected via the sensor of an Orchestrating AM.

3. **Replication with Validation** – The Orchestrating AM initiates the replication with validation strategy (3a.) by ensuring that the managed resource and the copy are identical with respect to their structure and behavior. A request is sent to the Orchestrating TM to set up the validation process (3b.), which loads the validation policy (3c.) into a Touchpoint TM.
4. Implement Change Plan on Copy – The Touchpoint AM implements the self-
management change plan on a readily available copy of the managed resource. This event would be a signal to the Touchpoint TM that self-testing can now be performed using the changed copy.

5. Ready for Self-Test – The Touchpoint TM detects that the resource is in a fully-adapted safe state, and is therefore ready to be tested.

6. Execute Test Cases – After analysis and planning, the Touchpoint TM executes test cases on the resource.

7. Log Test Results – The test results and coverage are coalesced into a log file, which is analyzed by the Touchpoint TM with respect to the validation policy.

8. Result of Post-Test Evaluation – Message indicating whether validation passed/-
failed, or if there was adequate test coverage, is sent to the Orchestrating TM.

9. Validation Response – Orchestrating AM is notified as to whether it should accept or reject the change request. If the change request is accepted, the Touchpoint AM implements the change on the actual managed resource; otherwise the change request is discarded.

Performing self-testing using resource copies means that validation can occur on a separate computational node. Therefore, in the presence of fast communication links between hardware nodes, RV leads to a highly transparent testing process. This is because: (1) regular service of the autonomic software need not be interrupted to perform validation; and (2) processing and memory overhead is distributed to additional hardware. The major limitations of the RV strategy are that it requires additional hardware and communication links; and is only applicable when managed resources can be replicated.
3.2.2 Safe Adaptation With Validation (SAV)

SAV tests self-management changes in-place, directly on managed resources, during system execution. Since testing must occur while managed resources are operating, the SAV strategy harnesses a disciplined approach to adaptation known as safe adaptation\(^{17}\) [79]. In order to apply this strategy, developers must implement the data structures and functions necessary for safe adaptation. However, SAV guarantees that neither validation nor adaptation will leave the system in an inconsistent state.

Under SAV, the steps of the workflow in Figure 3.4 differ from the RV strategy as follows:

3. Safe Adaptation with Validation – The Orchestrating AM initiates safe adaptation of the managed resource (3a) and concurrently requests that an Orchestrating TM set up validation (3b). An appropriate validation policy (3c) is then loaded into the test knowledge component of a Touchpoint TM.

4. Implement Change Plan on Managed Resource – The Touchpoint AM proceeds with safe adaptation until the managed resource is fully-adapted (i.e., up to Step 2 of the safe adaptation process outlined in Subsection 2.1.1) but keeps the resource blocked until validation is performed.

SAV offers an alternative to RV if it is too expensive or impractical to replicate managed resources. However, employing SAV can have adverse effects on the timing, processing, and memory usage of the autonomic software sources. Processing and memory usage drawbacks can be mitigated by raising the minimum system requirements for the software. However, timing still presents a significant drawback as some application services may have to be temporarily blocked while self-testing is performed. Therefore, SAV may not be a viable solution for real-time or mission-

\(^{17}\)Refer to Section 2.1.1 – “Ensuring Safeness” for background on safe adaptation
critical aspects of the software system. A state machine design to support SAV is provided in Chapter 4, Section 4.3.

Table 3.1 summarizes and compares the Replication with Validation, and Safe Adaptation with Validation strategies.

<table>
<thead>
<tr>
<th>Replication with Validation</th>
<th>Safe Adaptation with Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Can Occur on Separate Node</td>
<td>Testing Must Occur on Same Node</td>
</tr>
<tr>
<td>Highly Transparent Runtime Process</td>
<td>May Cause Performance Degradation</td>
</tr>
<tr>
<td>Does Not Interrupt System Services</td>
<td>Partially Blocks Some System Services</td>
</tr>
<tr>
<td>Requires Additional Hardware and Communication Links</td>
<td>No Additional Hardware Required</td>
</tr>
<tr>
<td>Requires Copies of Managed Resources</td>
<td>Does Not Require Resource Copies</td>
</tr>
</tbody>
</table>

Table 3.1: Comparative Summary of the RV and SAV Validation Strategies
CHAPTER 4

DESIGN METHODOLOGY TO SUPPORT SELF-TESTING

A detailed design methodology is used to support the seamless integration of self-testing into autonomic software. The methodology consists of: (1) an architectural view of autonomic software allows developers to identify which components should be made self-testable; (2) an illustrative example that demonstrates how global self-test coordination using the architecture maps to traditional\textsuperscript{18} forms of software testing; (3) a state-based model to support the realization of the SAV strategy within the architecture; (4) detailed designs of the major sub-components of the architecture, including automated support for three categories of software testing tools. The guidelines and rationale for design decisions thread the entire chapter.

Autonomic software may be viewed as a set of composable, interacting autonomic components that provide localized self-management through sensors and effectors, while being environmentally aware and connected via a system-wide control loop [72]. The component-based perspective of autonomic software is widely accepted and cited in the literature [48, 56]. Furthermore, the increasing popularity of Service-Oriented Architecture (SOA) [29], Web and Grid Services [1], and Cloud Computing [51], suggests that the component-based paradigm represents a pragmatic approach to building autonomic software systems.

In this chapter, a new component-based perspective of autonomic software is presented. Adaptive portions of the autonomic software are viewed as a set of \textit{Self-Testable Autonomic Components} (STACs). The boundaries of adaptation are delineated by analyzing the dependency relationships between managed resources and other software components. Any \textit{internal}\textsuperscript{19} software components that use the an

\textsuperscript{18}Unit, integration, system, and firewall regression testing

\textsuperscript{19}External software components are considered to be users that lie outside the system boundary
adaptive managed resource are within the firewall of change, and should therefore be made self-testable.

This chapter is organized as follows: Section 4.1 overviews the system-wide perspective of autonomic software as a collection of interconnected STACs. Section 4.2 describes the illustrative example of global STAC coordination. Section 4.3 provides the state-based safeness model. Section 4.4 formally defines the main elements of STACs. Section 4.5 outlines design goals to facilitate the development of STACs. Finally, Section 4.6 presents a set of detailed UML designs for STAC elements.

4.1 Architectural Perspective

Figure 4.1 presents a logical, component-based view of self-testable autonomic software using STACs. As shown at the center of Figure 4.1, STACs encapsulate the dynamically adaptive, managed resources of the autonomic software system, depicted by the boxes labeled A, B, C, E, and G. Recall that in the scope of the research problem being addressed, these managed resources represent mainstream software components that provide services associated with the application domain, e.g., banking, healthcare. Therefore, the arcs labeled test in Figure 4.1 represent the ability of each STAC to perform runtime testing on the domain services of the application.

At runtime, any of the mainstream components can be replaced by a component from the set of registered components (bottom-left of Figure 4.1), as long as the two components conform to the same service interface. If an autonomic change introduces a component that has not been previously tested within the application, self-testing is performed as part of its integration into the system.

Validation of the introduced component may require the use of stubs for any external dependencies (bottom-right of Figure 4.1. Stubs are created during the testing phase of the software development process, and should therefore be available at deployment time. However, high development costs, and the use of commercial-off-the-shelf (COTS) components, may make it impractical to deploy completely stubbed
Figure 4.1: STAC-Based Perspective of Adaptive Autonomic Software

replicas of the components with the system. Therefore, to incorporate the proposed
self-testing approach, the autonomic software design will facilitate: (1) redirecting
function calls to stubs at the class/object level, and (2) injecting stubs into compo­
nents at runtime.

The workflow of Figure 4.1 is described as follows:

- An autonomic manager (AM) continuously monitors the system environment
  (1.0) and analyzes its state according to a predefined adaptation policy.

- In response to environmental changes, the AM selects the replacement compo­
nents (2.0), and dynamically reconfigures the system (2.1).

- If runtime validation is required, a test manager (TM) intercepts adaptation
  and generates a test plan (3.0).

- The TM then requests that the AM set up validation (4.0) using any required
  stubs; and coordinates the testing process (4.1).
4.2 System-Wide Test Coordination

During the testing process (flow 4.1 in Figure 4.1), the Orchestrating TM directs a series of software adaptations and self-tests to validate the changed component(s). These coordinating activities can be divided into three phases: unit test phase, integration test phase, and firewall test phase. The detailed steps of the test coordination phases for each newly introduced component \( C_i \) are as follows:

**Unit Test Phase**

1. Replace the mainstream component targeted in the adaptation with \( C_i \).
2. If \( C_i \) calls other components, replace the callees with stubbed components; and invoke a self-test on \( C_i \).

**Integration Test Phase**

3. Generate an integration test order (ITO) for \( C_i \) to minimize the number of stub re-configurations required during \( C_i \)'s integration.
4. Use the ITO from Step 3 to invoke a series of adaptations and self-tests until all of the previously replaced callee components have been re-integrated.

**Firewall Test Phase**

5. Perform dependency analysis to identify the components that call \( C_i \); and invoke self-tests on them in reverse topological order.

**Illustrative Example.** Consider a scenario in which component E in STAC4, center of Figure 4.1, will be replaced by a new component D. Applying the steps for self-test coordination yields the following actions (depicted graphically in Figure 4.2):
Action 1. Replace E in STAC4 with D
Action 2. Replace G in STAC5 with S5
Action 3. Execute STAC4.selftest()  // End of Unit Tests
Action 4. Replace S5 in STAC5 with G
Action 5. Execute STAC4.selftest()  // End of Integration Tests
Action 6. Execute STAC3.selftest()
Action 7. Execute STAC1.selftest()  // End of Firewall Tests

Actions 1–3 set up and execute unit tests on D using the stubbed component S5; as depicted by the arc labeled Unit and the stereotype <<stubbed>> in Figure 4.2(a). Actions 4 and 5 correspond to the integration test shown in Figure 4.2(b), which validates D using its actual dependency G. Lastly, Actions 6 and 7 validate each caller component affected by D’s integration; as indicated by the arcs labeled Firewall in Figure 4.2(c). The stereotype <<blocked>> in Figure 4.2 indicates the partial disablement of caller components to ensure safeness. Recall from Section
Figure 4.3: State Machine of Safe Adaptation with Validation using STACs

3.2.2 that such blocking is necessary when testing is performed in-place during system execution, i.e., Safe Adaptation with Validation (SAV).

4.3 State-Based Safeness Model

Figure 4.3 shows the states and guarded transitions of the autonomic software system when STACs are used to implement SAV. It extends the work by Zhang et. al [79] by incorporating: (1) test setup activities through a refinement of the resetting, safe, and adapted states; and (2) test invocation, execution, and evaluation activities by adding a new validating state along with the appropriate transitions.

Before the safe adaptation with validation process begins, the system is in the running state where all components are fully operational. The Orchestrating TM then sends a reset command to the Orchestrating AM, which moves the system into the resetting state. During the reset, the system is in partial operation as the Orchestrating AM disables functions associated with the adaptation target AT, and
stubbed dependencies SD. Upon completion, the system is considered to be in a safe state where it is adapting for unit tests.

Once the unit test configuration is reached, the system transitions to the ready for unit tests state. The Orchestrating TM then sends a test command and the system enters the validating state, where test cases are executed. If unit testing is successful and the system has not yet reached its final configuration, the system moves to the adapting for integration tests state. The Orchestrating TM then issues another test command move the system back into the validating state. Runtime validation proceeds in this manner until the system reached its final configuration.

With the new component fully integrated, the system transitions from validating to the ready for firewall test state. The Orchestrating TM then invokes the final test command. If the firewall test passes, the adaptive change is accepted and the system enters to the resuming state. During resumption, the AT and SD functions are unblocked until the system returns the fully operational running state. At any point during unit, integration, or firewall testing the system Orchestrating TM can reject the autonomic change, and rollback to the previous configuration; as depicted by the dotted transition in Figure 4.3.

4.4 Formal Definition

Figure 4.4 provides the formal definiton of a STAC. The elements that make up this definition \((R, A, T, I, K)\) are also depicted graphically in Figure 4.5. Recall that the resource \(R\) represents a mainstream software component that is being managed by the autonomic system. \(R\) therefore provides domain services through the public interface \(I_S\). \(R\) can also be accessed through private interfaces for local autonomic management \((I_A)\) and runtime testing \((I_T)\), respectively. \(I_A\) is the uniform sensor and effector interface that allows autonomic managers \((A)\) to conduct self-configuration, self-optimization, self-protection, and self-healing activities. The interface \(I_T\) facili-
A self-testable autonomic component is defined as a 5-tuple $(R, A, T, I, K)$ where:

1. $R$ is a computational or informational software resource that provides a set of related services to its clients.
2. $A$ is a finite set of autonomic managers responsible for conducting self-management activities on $R$.
3. $T$ is a finite set of test managers, disjoint from $A$, for validating self-management changes to $R$.
4. $I$ is a finite set of interfaces for accessing client services ($I_s$), autonomic management ($I_a$), testing ($I_T$), and maintenance facilities ($I_M$).
5. $K$ is a knowledge source containing $R$’s test cases ($K_T$), validation policy ($K_P$), and test history ($K_H$).

Figure 4.4: Formal Definition of a STAC

Figure 4.5: Architectural Elements of a STAC

tates executing test cases on $R$ and gathering the results, and is supported on the backend by automated testing tools.

At the heart of the STAC is a knowledge source $K$, center of Figure 4.5, that is responsible for storing, organizing, and generating pertinent information on the component. This information includes: (1) Policies – rules that govern the autonomic behavior and testing activities of managers within the STAC, labeled $P_A$ and $P_T$ in Figure 4.5; (2) Test Cases – a comprehensive set of tests developed for the resource $R$, which may be executed and updated by some test manager $t \in T$; and (3) Test Histories – logs containing the results of testing over a specified time period. A maintenance interface labeled $I_M$ at the top of Figure 4.5 provides access to $K$. This
access may be in the form of internal calls from the autonomic or test managers, or external calls from the human administrator or other STACs. For example, a human administrator can manually update policies through $I_M$, or another STAC can query the component’s test logs to determine the results of testing.

4.5 Object Design Goals

The following goals have been identified to facilitate the detailed object design of STAC elements:

- **Genericity** – The resource $R$ should be templated so that it can be passed in and out of the STAC statically or dynamically, regardless of its data type. This supports integration with existing code, dynamic binding, and runtime testing.

- **Uniformity** – The autonomic interface $I_A$ and testing interface $I_T$ should be standardized to allow managers to access the resource in a uniform manner. Similarly, the maintenance interface $I_M$ of different STACs within the autonomic system should also be standardized.

- **Visibility** – The state of resource $R$ should be observable through $I_A$ and $I_T$ so that: (a) autonomic managers can determine when the resource requires self-management; and (b) test managers can verify state-based test results.

- **Controllability** – The state of resource $R$ should be controllable through $I_A$ and $I_T$ so that: (a) autonomic managers can implement self-management changes; and (b) test managers can force the resource into a specific pre-test state before applying test inputs.

- **Composability** – Each component should be dynamically composable with other STACs via the service interface $I_S$. We reference the definition of composition presented in [48, §3.3.1], which is similar to composition as defined by many component-based frameworks.
<table>
<thead>
<tr>
<th>Design Goal</th>
<th>Autonomic Benefit</th>
<th>Testing Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genericity</td>
<td>Adapting Components</td>
<td>Injecting Test Stubs</td>
</tr>
<tr>
<td>Uniformity</td>
<td>Using and Developing Autonomic Interface</td>
<td>Using and Developing Test Interface</td>
</tr>
<tr>
<td>Visibility</td>
<td>Detecting Symptoms</td>
<td>Verifying Postconditions</td>
</tr>
<tr>
<td>Controllability</td>
<td>Implementing Changes</td>
<td>Setting Up Preconditions</td>
</tr>
<tr>
<td>Safety</td>
<td>Blocking Components during Adaptive Changes</td>
<td>Intercepting AMs to Validate Changes</td>
</tr>
<tr>
<td>Reusability</td>
<td>Reducing Development Effort and Runtime Overhead</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Design Benefits to Autonomic and Testing Infrastructures

- Safety – The dynamic nature of the resource $R$, coupled with its multi-interface design, means that STACs should employ local and global access control mechanisms to ensure the consistency and integrity of $R$'s operations. In addition, TMs will need to be able to safely intercept AMs when validating changes.

- Reusability – A good design for STACs should facilitate the following forms of software reuse to limit software development costs and performance overhead: *Code*, reusing classes or collections of procedures and functions; *Interface*, reusing method signatures; and *Pattern*, reusing solutions to known problems.

Table 4.1 summarizes the benefits of the aforementioned design goal with respect to the development of the autonomic and testing infrastructures. In this research, special importance was placed on the reusability goal in an attempt to curtail adverse effects on the development effort, and runtime performance.

4.6 Detailed Object Design

This section presents a detailed design for STACs, including guidelines for specifying managed resources, autonomic and test managers, and manageability interfaces. The
4.6.1 Managed Resources

The design of the managed resource $R$ considers two scenarios: (1) developers have full access to create and modify $R$'s source code; and (2) $R$'s source code is inaccessible because it is commercial-off-the-shelf (COTS) software. These two scenarios impact the ability to provide autonomic services for $R$, and the degree to which $R$ can be designed for testability.

Figure 4.6 provides a detailed class design for $R$ when there are no limitations with respect to source code accessibility. Each managed resource derives from the class labeled \texttt{AbstractResource} in Figure 4.6. This abstract class provides safeness mechanisms at the resource-level to partially address the safety goal\textsuperscript{20} in Section 4.5. The safety mechanisms available to managed resources include:

- \texttt{Suspend()} – saves the current state and temporarily blocks all read/write access to the resource;

\textsuperscript{20}Manager-level safety is discussed in Section 4.6.2
• **Resume()** – restores the previous state and unblocks access to the resource;

• **SaveState()** – used by the suspend function to backup the current values of the resource’s attributes;

• **RestoreState()** – used by the resume function to recover previous values of the resource’s attributes;

State visibility and controllability are addressed by requiring that *getters* (accessor methods) and *setters* (mutator methods) be provided for all of $R$’s private and protected variables. These include variables of both primitive (built-in) types and non-primitive (abstract) data types.

The intrusiveness of the getter and setter requirement is more related to testing than self-management. For self-management, only the portion of the $R$’s state associated with its governing autonomic features needs to be visible and controllable. However, testing generally mandates full control of an object’s state to adequately setup the preconditions and verify the postconditions of test cases [10].

Another aspect of $R$’s design that was influenced by testability relates to the declaration of non-primitive data types. All non-primitive data declarations within $R$ should be specified as references to interfaces rather than implementation classes. For example, in Figure 4.6 object $A$ and object $B$ are declared in terms of the two corresponding interfaces Interface$A$ and Interface$B$, rather than concrete classes $A$ and $B$. The idea is that each object within $R$ should only know about its dependencies by their interface, thereby allowing different implementations to be swapped in/out transparently at runtime – a technique known as Dependency Injection (DI) [73]. DI leads to loose coupling and permits mock object implementations (i.e., test stubs) to be easily inserted into $R$ during runtime testing.

In the scenario where $R$ is COTS component, the adapter design pattern [33] can be used to provide the safety mechanisms defined in Figure 4.6. This involves building a wrapper class that adds suspend and resume operations to the component’s existing
functionality. Since R's source code is unavailable, its visibility and controllability are limited to the operations provided by the public interface.

Using COTS can impede both the manageability and testability of R when there is no way to break encapsulation, and gain access to the component's private or protected members. However, the complementary strategy presented in [9] can be used to improve testability of components when source code is not available. This is achieved by injecting the built-in test mechanisms directly into intermediate code, thereby enhancing the runtime testability of COTS components.

4.6.2 Autonomic and Test Managers

The proposed design for the set of autonomic managers (A) and the set of test managers (T) is shown Figure 4.7. Both types of managers use a generic design that extends the preliminary work in King et. al [45] by: (1) re-factoring common logic out of the monitor, analyze, plan, execute functions; (2) adding support for updating the internal knowledge of managers via an external knowledge source, and (3) incorporating safety mechanisms for suspending and resuming managers.

The GenericManager class, shown at the top-right of Figure 4.7, is the main controller class that coordinates the manager's activities. The generic manager may invoke an internal knowledge component or an external knowledge source; as indicated by the dependencies on the interfaces InternalKnowledge and KnowledgeSource respectively. Some key operations provided by the generic manager include:

- **Activate(p)** – sets the specified behavioral policy p as being active in the internal knowledge of the manager;
- **Manage()** – starts or resumes the self-management or self-testing activities of the manager;
- **Suspend()** – temporarily pauses the activities of the manager for control and safety purposes;
Figure 4.7: Generic Design of Autonomic and Test Managers

- **Update()** – retrieves new or updated behavioral policies from the external knowledge source.

The template parameter Touchpoint, represented by the dotted box at the top-right of Figure 4.7, is a place-holder for the class that implements the management or testing interface used by the manager. Therefore, upon instantiation the **GenericManager** requires the fully qualified class name of this Touchpoint class, and the name of the sensor method that will be used to poll the managed resource. The next subsection describes how autonomic control loops are realized within the generic manager using an OO implementation of the monitor, analyze, plan, execute (MAPE) functions.

**Abstract MAPE Functions**

The package `edu.fiu.strg.STAC.manager.mape` in Figure 4.7 shows the detailed class design of the MAPE functions of the manager. Each function derives from the abstract class **AbstractFunction**, which implements common behaviors such as: (1) independent initialization, suspension and resumption of the function; and (2) access to data shared among the functions.
Function independence is achieved through programming language support for multi-threading, as indicated by specialization of the Thread library (top-left of Figure 4.7). Once a MAPE object is initialized, its control thread continuously invokes a function-specific implementation of the abstract method doFunction(). For example, a monitor’s doFunction implements state polling of managed resources, while an analyze doFunction compares that state against known symptoms.

The boolean variable suspended in the AbstractFunction class denotes whether or not the function has been temporarily paused for safety purposes. Access to data shared by the functions is through the variable iKnowledge, which is a reference to an implementation of the InternalKnowledge interface.

Concrete MAPE Functions

The concrete MAPE is represented by classes labeled Monitor, Analyzer, Planner, and Executer shown in Figure 4.7. Both the Monitor and Executer classes incorporate reflection-oriented programming techniques [4] to avoid hard-coding the qualified method names of sensors and effectors. Therefore, at runtime class and method names can be modified to facilitate the management of adaptive resources.

All MAPE threads within the manager can be manipulated as a single unit. The class labeled MAPEGroup in Figure 4.7 contains the synchronization logic for initializing, suspending, and resuming a collection of MAPE threads. This may be supported in part by thread programming libraries, as indicated by inheritance from ThreadGroup.

Internal Knowledge of Managers

The internal knowledge of the manager is represented by the class KnowledgeData in Figure 4.8. As shown at the top of Figure 4.8, the KnowledgeData class conforms to the interface labeled InternalKnowledge, which is used by the MAPE functions of
the generic manager to access shared data. The key attributes of the KnowledgeData class are:

- **TouchData** – holds the current state information of the resource \( R \) (i.e., a mapping of \( R \)'s variables to their data values at a specified moment in time) captured by the monitor function, e.g., \( \{ x \mapsto 9 \} \), the variable \( x \) has a current value of 9.
- **Symptoms** – represents a set of conditional relations used by the analyze function to determine if \( R \) is in an undesirable state, e.g., \( x < 10 \), when the value of \( x \) is less than 10.
- **ChangePlans** – contains an action, or sequence of actions, generated by the plan function in order to transition \( R \) back to a desired state, e.g., \( \text{increment}(x) \), where \( \text{increment} \) is a function that increases \( x \)'s value by 1.

Combining the previous examples produces conditions necessary to invoke an autonomic control loop of the form: \( \text{while } x < 10 \ { x = x + 1; } \). Although a trivial example is presented, the proposed knowledge design facilitates the definition of complex autonomic behavior. This is achieved by allowing the state variables, operators, source of desired values, and sequence of actions to be specified as calls to user-defined functions.

**Illustrative Example.** Consider the following functions in a system: \( \text{memfree()} \) – returns the amount of free memory; \( \text{threshold()} \) – returns the minimum desired level of free memory; and \( \text{dealloc()} \) – releases allocated memory that is no longer in use when the data being held there is no longer needed. Using these functions, we can specify self-managing memory deallocation as the control loop: \( \text{while memfree()} < \text{threshold()} \ { \text{dealloc();} } \). However, the value referenced by \( \text{threshold()} \) can also be modified via the execution of other control loops, thereby facilitating the specification of non-trivial autonomic behavior.
Another aspect of our internal knowledge design that supports the specification of non-trivial autonomic behavior is related to the definition of symptoms, i.e., undesirable state conditions. Individual symptoms may be defined as the conjunction of relations between different state variables and their desired values. As shown at the left of Figure 4.8, the Symptom class is composed of one or more StateMapping objects. Each StateMapping consists of: (1) methods for accessing single state variable and its desired value, varAccess and valAccess respectively; and (2) a binary operator. The method isSatisfied() of the Symptom class is used to compute the truth value of each StateMapping, and return the logical conjunction of the results.

Satisfaction of a symptom triggers the generation of a ChangeRequest object, as depicted at the bottom of Figure 4.8. Change requests contain a symptom identifier sid that is used to lookup a ChangePlan that addresses the problem. Each ChangePlan is composed of one or more Action objects, which provides the name of the effector method and an optional Parameter list. Two queues, requestQueue and planQueue, have been incorporated into the KnowledgeData class to hold change requests and change plans, respectively. These blocking queues use producer-consumer relationships to facilitate safe communication among the MAPE functions in the
control loop. Safeness is also enforced through the use of the boolean variables `dataLocked` and `suspended`, which control access to shared data and facilitate function synchronization, respectively.

4.6.3 Management and Testing Interfaces

Recall that the set of interfaces \( \mathcal{I} \) to a STAC include: a client services interface \( \mathcal{I}_S \), an autonomic management interface \( \mathcal{I}_A \), a runtime testing interface \( \mathcal{I}_T \), and a maintenance interface \( \mathcal{I}_M \). However, since \( \mathcal{I}_S \) is application-specific and \( \mathcal{I}_M \) and is outside the scope of the research problem, only the design of \( \mathcal{I}_A \) and \( \mathcal{I}_T \) are discussed.

Autonomic Interface (\( \mathcal{I}_A \))

The design of the autonomic interface \( \mathcal{I}_A \) encapsulates sensors and effectors in a class that carries a reference to the resource \( R \). Sensor and effector methods use \( R \)'s object reference in their implementation to realize state monitoring, and the execution of low-level management tasks. This section presents the design of the concrete class that is passed to the generic manager as the template parameter `Touchpoint` in Figure 4.7. As shown in Figure 4.9, the `ConcreteTouchpoint` inherits from the abstract class `AbstractTouchpoint`, which provides access to the protected attribute `resource`. Each `sensorMethod` and `effectorMethod` is then implemented using the getters and setters of `resource`; as indicated by the method signatures in Figure 4.9.

Designing \( \mathcal{I}_A \) in this manner allows management services to be specified at the same level of abstraction as the management policies. For example, recall the symptom from the memory deallocation example in the previous section:

\[
\{ \text{while memfree()} < \text{threshold()} \}\]

When the desired threshold and \( R \)'s accessor method for available memory are both specified using absolute values (e.g., 256K), the sensor implementation return \( R.\text{getFreeMemInBytes()} \) is adequate. However, if the administrator wishes to express
the threshold in the policy as a percentage of total memory, then additional computation is required, e.g., \( \frac{R.getFreeMemInBytes()}{R.getTotalMemInBytes()} \times 100 \).
Implementing user-defined sensor and effector methods hides these additional computations from system administrators, allowing the use of convenient abstractions such as \texttt{memfree()} regardless of underlying logic.

Test Interface (\( I_T \))

In order to be compatible with the generic manager, the testing interface \( I_T \) must also inherit from the \texttt{AbstractTouchpoint} class in Figure 4.9. The difference between \( I_A \) and \( I_T \) relates to the sensor and effector implementations. Sensor methods in \( I_T \) are defined in terms of the state boolean variables declared in one or more autonomic managers. These boolean variables are used to indicate when AMs generate change requests that require runtime validation. Effector methods in \( I_T \) invoke a special self-test support interface that integrates a cross section of automated testing tools.

The design of the self-test support interface combines three categories of automated testing tools: (1) \textit{Test Execution} – applies test cases to the resource; (2) \textit{Code Coverage Analysis} – calculates the percentage of source lines of code and branches executed during testing; and (3) \textit{Test Performance Measurement} – computes the elapsed time, in milliseconds, taken to execute tests.
Figure 4.10: Automated Testing Interface

Figure 4.10 provides a detailed class diagram of the automated tool support implementation. As shown at the top of Figure 4.10, the automated tool support provides three interfaces associated with the aforementioned testing tool categories. The ExecutionCollector interface provides common services for test execution including:

- **SetExecToolLib()** – sets the library path of the test execution tool;
- **SetAuxToolLib()** – sets the library path of an auxiliary test support tool;
- **CleanUp()** – restores any environmental settings changed during testing;
- **Compile()** – builds the source code of a given resource or test script;
- **Execute()** – applies test cases to a given resource.

Code coverage and performance analyzers are viewed as auxiliary tools, which support the test execution services. The CoverageCollector and PerformanceCollector interfaces therefore inherit the functionality of the ExecutionCollector. Specialized features are then added to each of the collector interfaces. For example, the CoverageCollector provides mechanisms for instrumenting source code and gen-
erating test coverage reports, while the PerformanceCollector implements methods for timing test executions.

The concrete automated tool support implementation is represented by the Self-TestSupport (bottom-left of Figure 4.10), which realizes one or more of the collector interfaces. The attribute testResults within the SelfTestSupport class represents a data structure used to store the results of testing. After testing completes, this data structure can be queried by TMs to gather information such as the number of test failures; total elapsed time for testing; and the percentage branch and statement coverage. The class labeled SourceMap is used to configure the self-test harness by providing the location of the resource, its source code, and test scripts. An example XML definition for the source map data structure is provided in Appendix A.4.
A case study involving three autonomic software applications was conducted to evaluate the proposed self-testing approach. Self-testing was integrated into three autonomic software applications by using the: workflow and algorithms defined in Chapter 3; guidelines and supporting design artifacts in Chapter 4; and technical assistance and support of the author of this research dissertation. Each application was implemented in the Java [64] programming language, by a distinct development team. A suite of experiments was created using the measurable objectives and evaluation criteria in Section 1.3.

This chapter is organized as follows: Section 5.1 provides a detailed description of the applications used in the study; Section 5.2 describes the experiments and setup environment; Section 5.3 contains the experimental results; Section 5.4 evaluates the findings of the study, and discusses the lessons learned, limitations and threats to validity of the research investigation.

5.1 Applications

The following autonomic software applications were used in the study:

- **Autonomic Container (ACT) [63]**
  Development Team: Ronald Stevens (Lead), and Brittany Parsons.

- **Autonomic Job Scheduler (AJS) [58]**
  Development Team: Alain Ramirez (Lead), and Barbara Morales-Quinones.

- **Communication Virtual Machine (CVM) [2, 25, 77]**
  Development Team: Andrew Allen (Lead), Yali Wu, and Frank Hernandez.

Each application was developed in Java 1.5 using the Eclipse SDK [64, 68], and the necessary plugins to support development and testing [26, 32].
5.1.1 Autonomic Container (ACT)

During an initial survey of autonomic computing research projects, an open-source prototype of autonomic software could not be located. Therefore, the preliminary work in King et. al [44] introduced the concept of an autonomic container to demonstrate the proposed self-testing approach. An autonomic container is a data structure that possesses one or more of the self-management characteristics defined in Section 2.1 [44]. The purpose of the autonomic container was to provide a simplistic, lightweight (i.e., small size and low complexity) implementation model of how autonomic software operates. The autonomic container application was then used to investigate the challenges of testing autonomic software at runtime.

Autonomic Features

Stevens et. al [63] presents the implementation details of a self-testing autonomic container (ACT), which realizes the Replication with Validation strategy. As shown in Figure 5.1, ACT implements a stack (top-left) that can perform dynamic self-configuration and self-testing. A public interface, labeled StackService, provides data storage services to users including push, pop, top, isEmpty, isFull and oper-
Figure 5.2: Self-Test Subsystem of ACT Prototype

At runtime, when the stack reaches 80% of its full capacity, the subsystem labeled **SelfConfig** re-configures the stack to hold more elements through the protected interface **StackManagement**. However, prior to implementing the change, the **SelfTest** subsystem executes a set of baseline regression tests on **StackCopy** using the newly specified structure.

External users of the stack are simulated via the **UserSimulation** subsystem, which makes calls to the **StackService** interface in two modes: **Random** and **Always-Push**. Random mode determines whether to push or pop data elements by generating randomized boolean values, and the AlwaysPush mode continuously pushes data elements onto the stack. In both modes, the data elements for push operations are random **java.lang.Integer** objects.

### Self-Test Features

Figure 5.2 shows the design of the **SelfTest** subsystem of the ACT prototype. The **AutoToolSupport** interface provides services instrumenting the stack copy, executing regression tests, and collecting the test results. A suite of 15 test cases were stored in the class labeled **StackTest**, which extends **TestCase** from JUnit [32]. Branch and line coverage were computed through the **ProjectData** class of Cobertura [26]; To set up validation, the **OrchestratingTM** would instrument the **StackCopy** class in Figure
5.1 using Main from Cobertura. The TouchpointTM would then invoke StackTest to validate any structural changes to the stack.

5.1.2 Autonomic Job Scheduler (AJS)

Ramirez et. al [58] applied the autonomic container in the context of a more complex problem, i.e., short-term job scheduling. In an operating system (OS), a short-term scheduler determines the order in which jobs in memory receive the attention of the processor [60]. Many job scheduling algorithms have been developed, which vary with respect to optimization of different aspects of the system (e.g., response time, throughput). Autonomic Job Scheduler (AJS) is an application that simulates the behavior and environment of a self-managing, short-term job scheduler for an OS. A pool of agents, representing the processors of the system, service jobs according to a high-level scheduling algorithm. AJS assumes that the OS has two distinct, goal-oriented modes: (1) Interactive – jobs in the request queue have been submitted by on-line users who are awaiting a response; and (2) Batch – jobs in the request queue have been collected over a period of time and do not require an immediate response.

Autonomic Features

The autonomic features of AJS (version 1.0) include self-configuration, self-optimization, and implicit self-testing. Requests are stored in a queue container which structurally re-configures in a similar fashion to the ACT prototype (i.e., 80% full, increase capacity). In addition, behavioral changes are realized through dynamic replacement of the scheduling algorithm according to predefined symptoms. The following scheduling algorithms were implemented: (1) Shortest Request Next (SRN) – service jobs in increasing order of required processing time; and (2) First-Come, First-Serve (FCFS) – service jobs in the order they arrive. Self-optimization is achieved by adjusting the number of processing agents in proportion to the workload; while
self-testing validates any structural and behavioral changes using copies of the request container and agent pool (i.e., Replication with Validation).

Figure 5.3 provides a static view of AJS, which is comprised of: (1) two managed resources labeled RequestContainer and RequestAgents; (2) test copies of the aforementioned managed resources labeled ContainerCopy and AgentsCopy; (3) touchpoint AMs for self-configuration and self-optimization, labeled RCSelfConfig and RASelfOptim; (4) an orchestrating AM, labeled OAMScheduler; (5) a knowledge source, labeled KSScheduler; (6) two touchpoint TMs, labeled RCSelfTest and RASelfTest; and (7) an orchestrating test manager, labeled OTMScheduler.

A dynamic view of the self-management activities in AJS is depicted in Figure 5.4. The following scenario describes the autonomic behavior of AJS in the context of Figure 5.4:

- The system starts in the interactive mode using the FCFS algorithm, and new batch requests have just been added to the request queue.

- OAMScheduler, RCSelfConfig and RASelfOptim are concurrently monitoring different aspects of the system.
• **RCSelfConfig** increases the capacity of the request container every time allocated space exceeds 80 percent full; and continuously updates **KSScheduler** with statistics on the currently queued job requests (e.g., job types).

• **RASelfOptim** expands and contracts the pool of active processing agents to ensure that on average 8 agents are available for every 10 queued job requests.

• **OAMScheduler** analyzes the request statistics in **KSScheduler**; detects that batch jobs have been queued; generates a change plan for switching to the SRN algorithm; and flags validation of the behavioral change. In addition, **OAMScheduler** analyzes the change requests generated in **RCSelfConfig** and **RASelfOptim**, and flags validation for adaptive changes.

**Self-Test Features**

Self-testing in AJS is supported by JUnit [32] for test execution, and Cobertura [26] for calculating line and branch coverage. The initial AJS test set consisted of 32 test cases developed for the request container and agent pool. A subsequent point release
of AJS (version 1.1) incorporated self-protection into the request container, and an additional 7 test cases. Lastly, the self-test engine consisted of Touchpoint TMs for each managed resource, and an Orchestrating TM for global test coordination.

5.1.3 Communication Virtual Machine (CVM)

Communication Virtual Machine (CVM) [25] is a platform for realizing user-centric communication models. CVM is the result of an ongoing collaboration between FIU SCIS and Miami Children’s Hospital, where the application domain is realization of healthcare communication models. CVM separates the concerns of communication modeling, synthesis, coordination, and the actual delivery of the communication services into self-contained layers. These layers (from top to bottom of Figure 5.5) are:

1. User Communication Interface (UCI) – a modeling environment for users to specify their communication requirements as schema;
2. Synthesis Engine (SE) – a suite of algorithms for transforming user communication schemas into an executable control scripts;

3. User-Centric Communication Middleware (UCM) – a handler that executes control scripts to coordinate the delivery of communication services independent of the network configuration;

4. Network Communication Broker (NCB) – a uniform API for the UCM that self-configures the underlying network devices and infrastructure to realize communication.

Allen et al. [2] leverage autonomic computing, and open-platform communication APIs, in the provision of a comprehensive set of communication services for CVM. Figure 5.6 shows the architecture of the NCB layer of CVM. Through a series of dynamic adaptations, NCB self-configures the CVM to use use multiple communication frameworks such as Skype [61], Smack [41], and Native NCB [77] (bottom-left). An NCB Management layer exposes the high-level communication services to its upper layer via a network independent API (top of Figure 5.6).

When a request for communication is received, it is stored in the knowledge source labeled CallQueuing (middle-left). A Communication Services Manager (CSM) monitors the call queue for the requests, and invokes the service interface of the managed resource layer (bottom-right). This service interface, labeled NCBBridge, provides access to communication network-level communication services made available through a number of open-platform APIs such as Skype [61], Smack [41], and Native [77].

The services provided to the CSM via these communication frameworks include:

- Creating and destroying communication connections;
- Adding and removing participants from a connection;
- Enabling and disabling different types of communication media in a connection.
The adapter design pattern [33] was used to provide a standardized interface to the NCB bridge for each communication framework with an open-platform API, e.g., the interfaces labeled SkypeAdapter and SmackAdapter in the managed resource layer. Using the adapters allows the NCB to be extensible for inclusion of future communication frameworks. This is indicated by the dotted package labeled NewFramework and interface labled NewAdapter in the managed resource layer.

Autonomic Features

A Touchpoint AM, labeled TAMCommunication, directly manages the communication frameworks and is responsible for adapting the underlying network and device configurations. An Orchestrating AM, labeled OAMCommunication, monitors the CSM and Touchpoint AM, and also analyzes the requests in the call queue to determine if self-configuration is required.

Requests for NCB self-management have two general forms: (1) a user’s explicit request for communication that is not supported by the current configuration, thereby
requiring self-configuration; and (2) the occurrence of other external event such as failure in a framework, thereby requiring self-healing. Asynchronous signals were used in conjunction with an external knowledge source, labeled \textit{KSCommunication}, to coordinate the actions of the CSM, Orchestrating TM, and Touchpoint TM.

Self-Test Features

Self-testing was incorporated into the NCB using the Safe Adaptation with Validation (SAV) strategy. The orchestrating manager \texttt{OAMCommunication} was therefore responsible for disabling caller functions in the CSM and Touchpoint AM to ensure safeness during adaptation. The following automated tools were used to support self-testing in NCB: JUnit [32], JUnitPerf [17], and Cobertura [26]. Furthermore, a total of 41 regression test cases were written for the NCB to validate self-configuration changes generated from user requests and environmental events.

5.1.4 Summary of Applications

Table 5.1 summarizes the autonomic and self-testing features of the applications used in the case study. Each row of the table consists of the following information: application name; general application area and domain; environment type; validation strategies used (VS); test set size (TS); number of test managers (TM); number of self-testable autonomic components (ST); and indicators the autonomic features of self-configuration (SC), self-optimization (SO), self-protection (SP), and self-healing (SH). For example, the first row of Table 5.1 indicates that the Autonomic Container application provides data access services for general purpose users in a simulated environment; implements the Replication with Validation strategy; contains 15 test cases, 2 TMs, 1 STAC; and incorporates dynamic self-configuration.
Table 5.1: Descriptive Summary of Applications used in the Case Study

5.2 Experimental Setup

This section describes the environmental set up for the experiments that were used to evaluate the autonomic self-testing approach, and its supporting design methodology. Experiments are categorized according to the effectiveness of applying the approach as relates to: (1) building supporting infrastructures and prototypes of self-testable autonomic software, (2) minimizing negative impacts on performance when performing low-transparency self-testing; and (3) facilitating the detection of faulty change requests and implementation-based test coverage of the prototypes.

5.2.1 Development

Experiment Set 1 – consists of integrating a cross-section of self-testing activities into the applications used in the study. The infrastructures for autonomic management and automated tool support were implemented in Java 1.5 using the Eclipse 3.2 SDK. Programmers used synchronized threads, blocking queues, generics, and reflection in Java to implement each prototype. The self-test features developed for each application in this set of experiments is as follows:

1. Autonomic Container (ACT)

   Features → { Test Execution, Line and Branch Coverage, Test Evaluation }
2. Autonomic Job Scheduler (AJS)

*Features* \(\rightarrow\) \{ Test Execution, Line and Branch Coverage, Test Evaluation \}

*Strategies* \(\rightarrow\) \{ Replication with Validation \}

*Artifacts* \(\rightarrow\) \{ Design, Implementation \}

3. Communication Virtual Machine (CVM)

*Features* \(\rightarrow\) \{ Test Execution, Line Coverage, Timed Runs, Test Evaluation \}

*Strategies* \(\rightarrow\) \{ Safe Adaptation with Validation \}

*Artifacts* \(\rightarrow\) \{ Design, Implementation \}

Size and complexity metrics of the software artifacts produced during implementation were generated using the Eclipse Metrics v1.3.6 [13] plugin. The following data was automatically exported from Metrics to XML format as an estimate of development overhead: (1) Lines of Code, (2) Number of Methods, (3) Number of Classes, and (4) McCabe Cyclomatic Complexity. A windows-based Intel(R) Core 2 Duo 2.4GHz PC with 4GB RAM was used to collect the development metrics.

5.2.2 Performance

Experiment Set 2 - consists of two performance tests involving the AJS and CVM prototypes. These performance tests are as follows:

1. Non-Self-Test AJS vs. High Transparency Self-Test AJS

   *Generated Metrics* \(\rightarrow\) \{ Thread Utilization, Memory Usage \}

2. Low Transparency CVM vs. Optimized Low Transparency CVM

   *Generated Metrics* \(\rightarrow\) \{ Thread Utilization \}

For each experiment in this set the two variants of each system were compared. Performance metrics were acquired via the Eclipse Test and Performance Tools Plat-
form (TPTP) [69] v4.5.2 plugin. Experiment runs for the AJS involved simulating identical sets of user actions on both variants to induce self-testing. However, the self-testing infrastructure resided on a different node to the core AJS system. This was achieved by implementing AJS as a distributed system using Java Remote Method Invocation (RMI) [65]. For the CVM experiment, a self-configuration from a Skype two-way AV call to a Smack three-way AV call was used to initiate self-testing. However, self-testing in CVM was interleaved with the core system on the same node. A timeout of 250 milliseconds was introduced into the MAPE functions as an attempt to optimize interleaving of the self-testing features.

In both experiments in this set, TPTP was used to instrument the applications before inducing the self-testing behavior. A detailed thread and timestamp analysis of the experimental runs were exported from TPTP to Comma Separated Value (CSV) format for comparison. All machines involved in this round of experiments were windows-based, webcam-enabled, Intel(R) Pentium 4 PCs with 2GB RAM or greater. Lastly, for the CVM self-test performance results were obtained from the call initiator node only.

5.2.3 Test Set Quality

Experiment Set 3 – consists of seeding faults and capturing various metrics to measure the effectiveness of the test sets that were built for ACT, AJS, and CVM. Recall that the research scope does not include investigating techniques for improving test effectiveness and adequacy. However, the following experiments have been described for completeness and to ensure that the empirical study is repeatable.

1. Functional Analysis on ACT, AJS, and CVM

   *Generated Metrics* → { Number of Test Failures }

2. Program Coverage on ACT, AJS, and CVM

   *Generated Metrics* → { % Line Coverage, % Branch Coverage }
3. Mutation Analysis on ACT, AJS, and CVM

*Generated Metrics → { Mutation Scores }*

4. Timed Test Performance on CVM

*Generated Metrics → { Elapsed Time (ms) }*

Experiments using mutation analysis involved seeding faults into change requests for managed resources to simulate the generation of incorrect structural and behavioral changes. Table 5.2 describes the mutants that were created for the ACT, AJS, and CVM prototypes. Each row of the table provides the following information: abbreviated application name; managed resource involved in the mutation; categories for the mutants; and number of mutants in each category. The mutants in categories with the prefix **M-** were manually created by the testing team for the application; while those prefixed with **R-** were generated automatically method-level mutation operators from MuJava 3 [49].

Runtime test sets for each adaptable managed resource were written in JUnit 3.8.1 [32]. Cobertura 1.9 [26] was used to instrument the managed resources during testing to calculate line and branch coverage. JUnitPerf 1.9 [17] was used for timing test runs. An XML report generated by Cobertura was parsed to extract the code coverage results. In cases where JUnit tests failed, the results for line and branch coverage were omitted from the final computation. Test results for each prototype were gathered using a windows-based, Intel Pentium 4 PC with 2GB RAM or greater.

5.3 Results of Experiments

This section contains the findings of the different sets of experimental runs, and have been classified accordingly.
<table>
<thead>
<tr>
<th>Application</th>
<th>Mutated Resource</th>
<th>Mutant Categories</th>
<th># Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>Stack</td>
<td>M1. Improper Capacity Increase</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R1. Random</td>
<td>7</td>
</tr>
<tr>
<td>AJS</td>
<td>RequestContainer</td>
<td>Reused Mutants from M1.</td>
<td>5</td>
</tr>
<tr>
<td>RequestAgents</td>
<td></td>
<td>M2. Improper Agent Enablement</td>
<td>3</td>
</tr>
<tr>
<td>RequestAgents</td>
<td></td>
<td>M3. Improper Agent Disablement</td>
<td>3</td>
</tr>
<tr>
<td>SchedAlgorithm</td>
<td></td>
<td>M4. Improper Job Ordering</td>
<td>2</td>
</tr>
<tr>
<td>SchedAlgorithm</td>
<td></td>
<td>M5. Invalid Algorithm Syntax</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2. Random</td>
<td>20</td>
</tr>
<tr>
<td>CVM</td>
<td>NCBAdapters</td>
<td>M6. Improper Initialization / Reset</td>
<td>5</td>
</tr>
<tr>
<td>NCBAdapters</td>
<td></td>
<td>M7. Improper Participant Identification</td>
<td>3</td>
</tr>
<tr>
<td>NCBAdapters</td>
<td></td>
<td>M8. Improper Session Identification</td>
<td>3</td>
</tr>
<tr>
<td>NCBAdapters</td>
<td></td>
<td>M9. Improper Media Re-Configuration</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3. Random</td>
<td>24</td>
</tr>
<tr>
<td><strong>Total Mutants</strong></td>
<td></td>
<td></td>
<td><strong>85</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Mutants Generated to Simulate Faulty Change Requests

5.3.1 Development

This subsection contains the results of Experiment Set 1, and is divided into findings on the infrastructural development and prototype development.

Infrastructure

Figure 5.7 provides a summary of the results obtained from building the autonomic infrastructure and testing tool support. Each column of the table at the bottom of the figure provides measurements for the two infrastructures based on *lines of code*, *number of classes and methods*, and *cyclomatic complexity*. The stacked bar chart in Figure 5.7 compares the relative development effort required to build each of these infrastructures. For example, as indicated by the leftmost bar in Figure
5.7, incorporating autonomic support accounted for \( \sim 62\% \) of total infrastructural development effort with respect to lines of code. The remaining 38% was dedicated to providing testing support.

Prototypes

Figures 5.8 and 5.9 present the development metrics collected from static analysis of the prototype implementations. The prototypes were compared at different stages of their development as depicted by the key at bottom left of Figures 5.8 and 5.9). The key has three main categories: (1) Non-Autonomic – prototype implementation before any autonomic features were introduced, (2) Autonomic – prototype implementation incorporating autonomic features; and (3) Autonomic-ST – prototype implementation incorporating both autonomic and self-test features. It must be noted that only the source lines of code (LOC) metric was used as the basis for prototype comparisons.

A stacked bar chart is provided for comparing the raw LOC data measurements for the autonomic container (ACT) v1.0, and autonomic job scheduler (AJS) versions 1.0 and 1.1 in Figure 5.8. These three prototypes were built incrementally as a means
Figure 5.8: Comparison of Incremental ACT and AJS Development Effort

of investigating the RV strategy as the software system expands. This allows the development of the prototypes to be compared both vertically (i.e., within the same version), and horizontally (i.e., across different point releases).

The bar chart in Figure 5.9 shows the relative development effort of building different aspects of the prototypes, including the NCB layer of CVM. As indicated by the right-most table column at the bottom of Figure 5.9, the non-autonomic version of the NCB is a medium-sized software component with a total of 6466 LOC. Introducing autonomic and self-testing capabilities into the NCB increased its size by 1817 SLOC and 441 SLOC, respectively. Due to the large size of the NCB in comparison to the other prototypes, the y-axis of the bar chart in Figure 5.9 measures the percentage of total LOC, rather than absolute values.
5.3.2 Runtime Performance

This subsection contains the results for Experimental Set 2, and is divided into findings on the performance of AJS and CVM.

Autonomic Job Scheduler

For the AJS performance experiment, the percentage difference between the non-self-test and high transparency self-test variants were < 1% for both thread utilization and memory usage.

Communication Virtual Machine

The performance results for the NCB layer of CVM is shown in Figure 5.10. Five sample runs of the NCB’s test management thread utilization were taken throughout the execution of the experiments, and are shown along the x-axis of Figure 5.10. The
y-axis of Figure 5.10 shows the thread utilization of the AMs and TMs corresponding to each of the sample runs.

5.3.3 Test Set Quality

This subsection contains the results of Experimental Set 3, and is divided into findings of the mutation analysis, code coverage analysis, and timed test executions.

Mutation Analysis

Table 5.3 contains the mutation scores for each application. Recall that the mutation score metric is the ratio of the number of mutants killed to the total number of mutants. The forth row of Table 5.3 expresses the mutation score for each application as a percentage.
<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Mutants Killed</th>
<th>Number of Total Mutants</th>
<th>Mutation Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>12</td>
<td>12</td>
<td>100</td>
</tr>
<tr>
<td>AJS</td>
<td>32</td>
<td>34</td>
<td>94.1</td>
</tr>
<tr>
<td>CVM</td>
<td>35</td>
<td>39</td>
<td>89.7</td>
</tr>
</tbody>
</table>

Table 5.3: Results of the Mutation Analysis Experiments

Code Coverage Analysis

Table 5.4 contains the line and branch coverage results for each application. Recall that branch coverage results were not computed for the CVM prototype and therefore is omitted from Table 5.4.

<table>
<thead>
<tr>
<th>Application</th>
<th>Line Coverage (%)</th>
<th>Branch Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>80</td>
<td>75</td>
</tr>
<tr>
<td>AJS</td>
<td>77</td>
<td>70</td>
</tr>
<tr>
<td>CVM</td>
<td>63</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5.4: Results of the Program Coverage Analysis Experiments

Timed Test Analysis

Table 5.5 contains the timed test results for the CVM prototype. Original time formats from JUnitPerf was in milliseconds, but were converted to seconds for readability.

<table>
<thead>
<tr>
<th>Application</th>
<th>Elapsed Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVM</td>
<td>213.3</td>
</tr>
</tbody>
</table>

Table 5.5: Results of the Timed Test Performance Analysis Experiments
5.4 Empirical Evaluation

The primary objective of conducting the case study was to evaluate the proposed self-testing approach, and the supporting design methodology. Preliminary investigations sought seamlessly integrate the notion of self-testing into an existing autonomic software product. However, when no open-source autonomic software products could be located, the research focus shifted to developing prototypes to determine the feasibility of Replication with Validation (RV).

Overlapping research ideas led to collaborations with the CVM development team. Applying the Safe Adaptation with Validation (SAV) strategy to CVM, provided a real, large-scale, autonomic software development platform for the investigation. This section describes the observations, challenges, and lessons learned from conducting the case study. In addition, threats to the validity of the results for our experimental data is discussed.

5.4.1 Observations

Aspects of the experimental results are highlighted to substantiate the use of autonomic self-testing. More specifically, evidence to support the established benefits to development and runtime performance is emphasized.

Autonomic Self-Test Development

The results on infrastructural development (Figure 5.7) reveal that on average it took 10-20% percent less effort to develop self-test support than the autonomic infrastructure. This proved to be true for each of the size metrics used in the evaluation (Lines of Code, Number of Methods, Number of Classes), and the complexity metric (McCabe Cyclomatic Complexity).

In the autonomic infrastructure, the function with the highest cyclomatic complexity number was the isSatisfied() method (1.733). This method was the one
which determined whether or not the managed resource was in a state requiring self-management. In the self-test infrastructure, the method with the highest cyclomatic complexity number was the `execute()` method (1.402). This method was used to apply test cases to the managed resource with the appropriate auxiliary tools enabled.

Analysis of the incremental prototype development in Figure 5.8 reveals additional evidence that the development overhead of self-testing was less than self-management. In all cases, even though there was a one-to-one ratio of AMs to TMs, there was significantly less code dedicated to implementing self-test features. This is because the sensor/effector methods, and resource code modifications for self-management were inherently more complex than their self-testing counterparts.

Comparing the ACT, AJS, and CVM LOC metrics in Figure 5.9 shows that self-test drivers took 15-25% less development effort than those for autonomic management. Developers also reported that the self-test policies for TMs were relatively easy to develop, and at most required formulating one or two boolean state mapping for symptom detection. On the other hand, some AM policies implemented as much as seven state mappings depending on the system requirements.

Runtime Performance

Performance results for AJS under Replication with Validation showed that this strategy provides a highly transparent testing process. Less than an 1% increase in processing and memory usage was experienced, and there were no visible signs of degradation. This is because the TMs monitored AMs remotely, and the cost of periodic remote method invocations to check the values of one or two boolean variables.

Introducing a small timeout into the TMs of the NCB significantly improved its runtime performance. As shown in Figure 5.10, without the timeout 65-85% of process interleaving was dedicated to AMs and TMs. Using such an approach there were high levels of degradation in the CVM with respect to establishing communication connections. Once the timeout in the MAPE was activated, the AMs and TMs only
utilized 20-35% of the process execution time. This placed the degradation of the CVM within acceptable bounds. However, there was a 250 millisecond trade off in the responsiveness of symptom and validation detection.

5.4.2 Lessons Learned

Synchronization between components and ensuring harmony between the closed control loops were the major challenges experienced when developing the ACT and AJS prototypes. Although ACT and AJS are relatively small autonomic systems, the heavy use of threads and lack of safety mechanisms made debugging them quite challenging. Feedback from the ACT and AJS prototypes therefore led to updating the initial manager design to include safety mechanisms. In addition, tests to address the adaptive changes within ACT and AJS were not very difficult to implement.

Using safety mechanisms to debug the self-testable NCB of CVM was instrumental to the success of the project. Prior to integrating self-testing, CVM was already heavily multi-threaded, and this complexity was compounded by the large size of the software. Developers reported that debugging and off-line testing of NCB relied heavily on the use of the embedded suspend and resume operations of the generic manager. Developing tests for the NCB also proved to be challenging. This was because some tests required asynchronous calls to the underlying communication frameworks. Most open-source testing tools do not support asynchronous testing, and hence CVM developers had to implement an addition test support program for this purpose.

Selecting and tailoring open-source testing tools for use in autonomic software also presented some difficulties. Some of the difficulties experienced in this regard were: (1) locating trustworthy information on tool features and configuration procedures during an initial survey; (2) filtering tools that would not be suitable because of additional dependencies, e.g., Apache ANT, (3) determining which report formats (XML, HTML, CSV) would be most suitable for obtaining metrics in a uniform
manner. Many of the command line testing tools had poor error reporting capabilities. In addition, the slightest deviation from their command line parameters would result in failure traces that were hard to decipher. Special care therefore had to be taken to ensure that all class path configurations were included, and ordered correctly, by the self-test support implementation.

5.4.3 Threats to Validity

No other empirical studies were found in the literature comparing the overhead of integrating runtime testing into self-managing software, and therefore it is difficult to verify the results of the case study. A major threat to the validity of the results is the inexperience of the autonomic development and testing teams. With the exception of the CVM team, many of the software engineers involved in building the prototypes were relatively new to the problem domain and field of AC. Furthermore, the lack of open-source AC projects meant that the developers did not have access to other detailed designs and implementations to guide their activities. The information provided by such artifacts would have reduced the difficulties in specifying the low-level details of AMs, MAPE functions, and knowledge sources.

Another possible threats to the validity of our results include not considering the time required to develop autonomic features and test cases irrespective of the size of the resultant source code or test scripts. For example, although the LOC metric for the test cases and test drivers for the CVM prototype was not very high, making the NCB self-testable took a significant amount of time because of the need for asynchronous test calls. This may have also been impacted by the inexperience of the testing team in developing test scenarios for a real-time, communication-intensive software application. Current limitations of the case study include lack of dynamic planning mechanisms, and test case generation.
CHAPTER 6
CONCLUSION

This chapter provides a summary of the research presented in this dissertation, and discusses future directions in the areas of autonomic computing and software testing. In the research summary, the results of the empirical study are compared with the evaluation criteria of the research objectives. This comparison is used as the basis for demonstrating how the proposed solution satisfies the research goal. The future work section describes potential research projects and investigations that will be based on autonomic self-testing. This final chapter is organized as follows: Section 6.1 summarizes the research; Section 6.2 describes the major contributions of the dissertation, and Section 6.3 discusses the future work.

6.1 Research Summary

The research problem identified in this dissertation was divided into the following sub-problems: (1) formulate an approach to facilitate runtime testing of adaptive self-management features in autonomic software; (2) refine the testing approach in the first sub-problem to address the need for different levels of runtime efficiency in autonomic software; (3) investigate ways in which any unavoidable degradation in performance imposed by the approach can be limited; and (4) develop a detailed design methodology to support autonomic software developers when implementing the approach.

Investigation into the first sub-problem led to the development of a new, implicit self-test feature for autonomic software; referred to as Autonomic Self-Testing. Chapter 3 presents the autonomic self-testing approach by describing: (1) the duties that will be carried out by autonomic managers specialized for runtime testing, referred to as test managers (TMs); (2) the local self-test actions and interactions of TMs when
validating self-management changes, including access to automated testing tools; (3) detailed algorithms for implementing the TM self-test actions and interactions as autonomic control loops; and (4) general steps for seamlessly integrating TMs into the workflow of standard autonomic managers. Descriptions 1–3 are presented in Section 3.1, and Description 4 is presented in Section 3.2.

Table 6.1 shows how the research presented in Chapter 3 satisfies the Objective 1 in Section 1.3. Recall that the criteria for Objective 1 required for existing autonomic software to be extended with runtime testing capabilities such as: test execution, code coverage analysis, timed test performance, and post-test evaluations. The requirement was that the design or implementations be able to perform three testing activities in the same design or implementation.

The rows of Table 6.1 demonstrates that these criteria were met by each application used in the case study. In addition, the autonomic self-testing is guaranteed to be 100% compatible with any autonomic software application, as the approach is implemented using the same infrastructure as the self-management subsystem. Furthermore, at least two of the applications used in the study implemented different levels of testing transparency; thereby demonstrating the flexibility of the approach with respect to addressing varied performance requirements.

The solution for the second sub-problem was provided in Section 3.2, which presents the workflow for autonomic self-testing in the context of two validation strategies – Replication with Validation (RV), and Safe Adaptation with Validation (SAV). The RV strategy targets autonomic software systems that have strict runtime
Table 6.2: Prototype Coverage of Research Objective 2

<table>
<thead>
<tr>
<th>Application</th>
<th>Thread Utilization Difference / Range (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJS { High Transparency }</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>CVM { Low Transparency }</td>
<td>20 to 35</td>
</tr>
</tbody>
</table>

performance requirements. This is achieved by shifting the majority of the runtime overhead\textsuperscript{21} to a separate computational node; thereby producing a highly transparent testing process. However, RV requires additional hardware, and mandates that the autonomic system maintain copies of managed resources.

SAV was developed as an alternative strategy to RV. The SAV strategy could be used if it was infeasible or impractical to replicate managed resources. However, such a strategy meant that validation had to be performed on the same computational node as the autonomic software. Therefore, SAV was recognized as being a visible runtime testing process because it could lead to observable signs of performance degradation. Formulating the SAV strategy allowed for investigation into the third sub-problem, which involved optimizing the performance overhead of visible testing processes. Performance issues relating to RV and SAV were studied through a set of practical experiments in Chapter 5.

Table 6.2 shows how the results of the performance experiments, presented in Section 5.2.2, accomplish Objective 2. Recall that the criteria for Objective 2 requires negligible\textsuperscript{22} differences in performance when using highly transparent self-testing. In addition, when using low transparency self-testing, the interleaving of testing process should be less than that of the core system. As shown by the AJS and CVM prototypes in Table 6.2, the performance experiments were more than favorable with respect to these research requirements.

\textsuperscript{21}Processing, memory, and timing

\textsuperscript{22}Defined in the research objectives as being less than 5%
Chapter 4 presents the findings of the fourth and final sub-problem that was investigated. It describes a detailed design methodology to support autonomic self-testing. The primary goal of the methodology is to limit the effort required to build self-testable autonomic software (either from the bottom-up or through its integration into existing systems).

The design methodology consists of: (1) an architectural view of autonomic software allows developers to identify which components should be made self-testable; (2) an illustrative example that demonstrates how global self-test coordination maps to traditional\(^23\) forms of software testing; (3) a state-based model to support the realization of the SAV strategy within the architecture; (4) detailed designs of the major sub-components of the architecture. Furthermore, the guidelines and rationale for design decisions are threaded throughout the chapter.

Table 6.3 shows the prototype coverage of Objective 3, which demonstrates fulfillment of the final research requirement. Recall that the criteria for Objective 3 mandates that the development effort for supporting and driving testing activities is strictly less than its self-management counterparts. Each row in Table 6.3 reflects the attainment of this final objective. In conclusion, the findings of the study reveal that autonomic self-testing provides a flexible approach for building safe, reliable autonomic software; while limiting the development and performance overhead of having an integrated testing process.

\(^{23}\text{Unit, integration, system, and firewall regression testing}\)
6.2 Contributions

This research dissertation establishes the following novel contributions in the area of autonomic computing:

1. Creation of an implicit self-test feature for validating dynamically adaptive, self-management changes in autonomic software. This includes the description of a workflow and algorithms for tailoring the existing autonomic infrastructure for testing; and the formulation of two validation strategies that achieve different levels of runtime testing transparency.

2. Definition of a new, logical, component-based perspective for autonomic software, which allows developers to focus on implementing self-management changes safely and reliably. This includes the specification of design goals, UML design schematics, policy formats, and guidelines that reduce the effort required to build self-testable autonomic software.

3. Provision and analysis of experimental data to support: extending of autonomic software with runtime testing; tailoring runtime testing performance to achieve different levels of transparency in autonomic software; and limiting the development effort required to integrate runtime testing into autonomic software. This includes the experiences and lessons learned from integrating a variety of runtime testing activities into these self-managing systems.

6.3 Future Work

The research presented in this dissertation provides the foundation for investigating various research directions that support dynamic quality assurance in autonomic software. One such research direction is the use of formal specifications to dynamically generate test sequences, test oracles, and test data for system components [7]. Embedding formal specifications within autonomic system components will provide
a means for extracting precise descriptions of useful test information such as preconditions, postconditions and invariants. Furthermore, executable formal specifications facilitate the development of visualization systems [27], which has been identified as one of the major research challenges of autonomic computing [42]. Such visualization systems can be used to interactively guide the administrator through test scenarios and during system debugging.

Future work also calls for the investigation of policy-based risk analysis and trust mechanisms [31] to better tackle the problem of testing interactions with new component services. Testing requirements will be based on measurements of the estimated risk of interactions with these newly integrated entities, or in the presence of unknown environmental conditions. Furthermore, a risk-based strategy to regression test selection [16] will be investigated to identify test cases which would be most appropriate for testing high-risk components.

Lastly, the reusable designs presented in Chapter 4 will be used as the basis for a Graphical Autonomic Modeling Environment (GAME). GAME will provide a framework for creating or importing software projects, and using a graphical canvas to specify the structure and behavior of self-managing and self-testing components. Model-driven software development techniques will be incorporated into GAME to allow developers to specify self-management and self-testing components as platform independent models, which will then be automatically translated into platform specific models for execution.
APPENDICES
APPENDIX A

POLICIES

A1. Touchpoint Self-Configuration XML Policy for ACT

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<policy name="TAM Self-Config" version="1.0">

<monitor touchpackage="edu.fiu.strg.ACT"
    touchclass = "TouchStack"
    sensormethod="getState"/>

<analyzer>
    <symptom sid="TAM_SC_001">
        <mapping var="getFullPercent"
            op = "geq"
            val="80" />
    </symptom>
</analyzer>

<planner>
    <plan pid="TAM_SC_001">
        <action effector="increaseCapacity"/>
    </plan>
</planner>

</policy>
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<policy name="OTM Self-Test" version="1.0">
  <monitor touchpackage="edu.fiu.strg.ACT"
    touchclass="OTMTouchData"
    sensormethod="getCoordState"/>
  <analyzer>
    <symptom sid="OTM_ST_001">
      <mapping var="requiresSC"
        op="eq"
        val="true" />
    </symptom>
  </analyzer>
  <planner>
    <plan pid="OTM_ST_001">
      <action effector="dequeueReqSC"/>
      <action effector="setupResource"/>
      <action effector="deployTouchTM"/>
    </plan>
  </planner>
</policy>
<?xml version="1.0" encoding="ISO-8859-1"?>
<policy name="TTM Self-Test" version="1.0">
    <monitor touchpackage="edu.fiu.strg.ACT"
             touchclass = "TTMTouchData"
             sensormethod="getState"/>
    <analyzer>
        <symptom sid="TTM_ST_001">
            <mapping var="readyForST"
                     op = "eq"
                     val="true" />
        </symptom>
        <symptom sid="TTM_ST_002">
            <mapping var="testComplete"
                     op = "eq"
                     val="true" />
        </symptom>
    </analyzer>
    <planner>
        <plan pid="TTM_TT_001">
            <action effector="executeTests"/>
        </plan>
        <plan pid="TTM_TT_002">
            <action effector="evaluateTests"/>
        </plan>
    </planner>
</policy>
<?xml version="1.0" encoding="ISO-8859-1"?>
<sourceMaps>
  <sources>
    <source name="JUnit"
      type="tool"
      path="/tools/junit4.5/junit-4.5.jar" />

    <source name="Cobertura"
      type="tool"
      path="/tools/cobertura-1.9" />

    <source name="CodeCover"
      type="tool"
      path="/tools/codecover.bat" />

    <source name="JUnitPerf"
      type="tool"
      path="/tools/jperf/lib/junitperf.jar" />

    <source name="JUnitPerf"
      type="supportingClasses"
      path="/tools/jperf/src/com/clarkware/junitperf" />

    <source name="src"
      type="classpath"
      path="/cvm/src" />

    <source name="SmackAdapter"
      type="resource"
      path="/cvm/src/ncb/SmackAdapter.java" />

    <source name="SmackTest"
      type="test"
      path="/cvm/src/ncb/SmackTest.java" />
  </sources>
</sourceMaps>
BIBLIOGRAPHY


[59] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pages 68–71, New York, NY, USA, 1996. ACM.


VITA

TARIQ M. KING

08/2009  Doctor of Philosophy in Computer Science
         Florida International University, Miami, Florida

08/2008  Instructor in Undergraduate Software Engineering
         Florida International University, Miami, Florida

05/2008  Researcher in Service Building Technology Department
         IBM China Research Lab, Beijing, P.R.C

04/2007  Master of Science in Computer Science
         Florida International University, Miami, Florida

08/2005  Research and Teaching Assistant in Software Engineering
         Florida International University, Miami, Florida

12/2003  Bachelor of Science in Computer Science
         Florida Institute of Technology, Melbourne, Florida

12/1999  Associate in Information Technology, Economics, Geography
         Barbados Community College, St. Michael, Barbados

07/1979  Born, St. Michael, Barbados

PUBLICATIONS AND PRESENTATIONS


Peter J. Clarke, Djuradj Babich, Tariq M. King, B.M. Golam Kibria: Analyzing Clusters of Class Characteristics in OO Applications. JSS 81(12) 2008: 2269-2286

Peter J. Clarke, Yali Wu, Andrew A. Allen, and Tariq M. King: *Experiences of Teaching Model-Driven Engineering in a Software Design Course.* MODELS 2009 Educators Symposium: October 6


Tariq M. King, Alain E. Ramirez, Barbara Morales, Peter J. Clarke: *A Reusable OO Design to Support Self-Testable Autonomic Software.* SAC 2008: 1664-1669


Tariq M. King, and Peter J. Clarke: *Practical Unit Testing for Instructors of CS1, CS2, CS3.* WISTPC 2009: March 16-17

Tariq M. King, Peter J. Clarke, and Jun Zhu: *Leveraging Traceability Recovery in Test Planning and Optimization.* LA Grid Summit 2008: October 30-31

Jairo Pava, Yanelis Hernandez, Tariq M. King, and Peter J. Clarke: *A Model-Driven Approach to Web Application Testing.* SRAI 2008: April 4
