A Self-Testing Autonomic Container

Ronald Stevens
Department of Computer and Information Sciences
Florida A & M University
Tallahassee, FL 32307
rstevens@cis.famu.edu

Brittany Parsons
Department of Mathematics, Computer Science and Physics
Capital University
Columbus, OH 43209
bparsons@capital.edu

Tariq M. King
School of Computing and Information Sciences
Florida International University
Miami, FL 33199
tking003@cis.fiu.edu

ABSTRACT
Many strategies have been proposed to address the problems associated with managing increasingly complex computing systems. IBM's Autonomic Computing (AC) paradigm is one such strategy that seeks to alleviate system administrators from many of the burdensome tasks associated with manually managing highly complex systems. Researchers have been heavily investigating many areas of AC systems but there remains a lack of development in the area of testing these systems at runtime. Dynamic self-configuration, self-healing, self-optimizing, and self-protecting features of autonomic systems require that validation be an integral part of these types of systems. In this paper we propose a methodology for testing AC systems at runtime using copies of managed resources. We realize the architecture of a self-testing framework using a small AC system. Our system is based on the concept of an autonomic container, which is a data structure that possesses autonomic characteristics and added ability to self-test.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *validation, reliability*

General Terms
Verification, Reliability, Management.

Keywords
Autonomic Computing, Validation, Testing.

1. INTRODUCTION
As businesses continue to expand, computing systems are becoming more complex and almost impossible for human administrators to manage. The challenge of managing these highly complex systems also places a burden on businesses with respect to overall cost and maintenance. The Autonomic Computing (AC) paradigm [7] was developed in an effort to address this problem. AC attempts to relieve human managers of the complex and costly maintenance associated with large-scale systems by embedding self-management capabilities within the system. The currently accepted structure of AC systems includes self-healing, self-protecting, self-configuring, and self-optimizing characteristics. Although AC is still in the early stages of development, there has been great emphasis in the research community due to the impact these systems can have on businesses with a large IT infrastructure. The preliminary work on this paper focused on identifying the areas of autonomic computing currently being researched. A literature survey revealed that one of the most dormant areas of AC research is the dynamic validation of these self-managing systems. Testing, one of the main approaches of validation, is an essential phase in the development of computing systems. The difficulty and lack of testing is a serious issue that must be addressed while working to develop large scale systems. This is particularly true for self-managing systems that have the ability to modify themselves at runtime. Therefore, thorough testing is required during the development of AC systems, but more importantly these systems must be able to perform runtime validation.

The main objective of this research is to introduce a framework for testing AC systems at runtime. We were unable to locate a freely available autonomic system to allow us to seamlessly integrate the notion of self-testing into its functionality, and hence we introduce the concept of an autonomic container. An autonomic container is a data structure that has self-managing capabilities, and also has the implicit ability to self-test. We do not implement all the characteristics of the autonomic container as our focus in this work is the self-testing capability.

In this paper, we present the proposed self-test ability of an autonomic system using our prototype of an Autonomic Container. Our approach integrates a self-test manager into the autonomic computing system framework [8]. The self-test manager allows changes made to an autonomic system to be validated at runtime, ensuring that the desired functionality within the system is correct. Our testing framework allows for automatic: (1) runtime testing of an autonomic system after a change has been made and (2) generation of test results after runtime testing has been performed. In this initial work, we have implemented a reduced version of an Autonomic Container and validated our testing framework using a prototype.

In the next section we give an overview of Autonomic Computing and describe common testing approaches. Section 3 introduces the concept of an autonomic container. Section 4 presents the layout of the architecture for our testing framework for autonomic containers. Section 5 describes our prototype that implements the autonomic testing framework. Section 6 provides the related work, and in Section 7 we conclude and describe plans for future work.

2. BACKGROUND

This section provides an overview of the Autonomic Computing paradigm as developed by IBM, summarizes some current AC research, and discusses the main approaches to software validation.

2.1 Overview of Autonomic Computing

Autonomic Computing is defined as an environment in which computers can manage themselves based on high-level specifications set forth by a system administrator [8]. While numerous architectures have been proposed, autonomic systems are generally accepted to have four key characteristics – self-configuration, self-protection, self-optimization, and self-healing.

The concept of self-managing computing systems is not a new one, but the term “Autonomic Computing” was coined in March 2001 by IBM’s research department [9]. The word *autonomic* is derived from the human autonomic nervous system. The human autonomic nervous system performs basic functions such as regulating the heartbeat without the need for conscious thought. Autonomic computing extends this concept to computing systems that can perform automatic self-management activities in a similar fashion. Additional low-level infrastructure is embedded in AC systems to automatically regulate the overall functioning of the system.

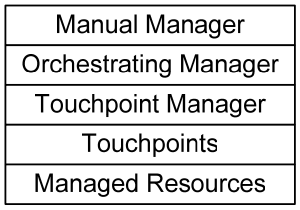

Figure 1: Layered architecture of an autonomic system

Figure 1 shows the IBM’s layered architecture for an AC system which consists of managed resources, touchpoints, touchpoint managers, orchestrating managers, and manual managers. The lowest and most important layer of the architecture is the managed resource layer. It is the layer in which the change resulting from self-management generally takes place. A managed resource can be any software or hardware that is manageable [8]. The basic building block of an autonomic system is an autonomic element as shown in Figure 2. Autonomic elements consist of a managed resource and a touchpoint autonomic manager, and are characterized by closed control loops. These closed control loops consist of a monitor stage, analyze stage, plan stage, and an execute stage. These four stages use a shared knowledge base to function according to high level policies. The touchpoint autonomic manager interacts with the managed element through a touchpoint interface consisting of sensors and effectors.

Autonomic Managers (AMs) all share a similar architecture. Each AM features an intelligent control loop that can monitor, analyze, plan, and execute events based on shared knowledge.

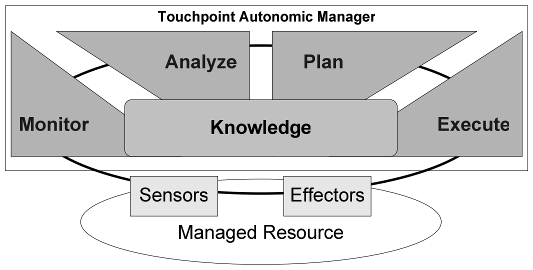

Figure 2: Autonomic Element Architecture

Managed resources are governed directly through their touchpoints by Touchpoint AMs.

Each autonomic characteristic of self-healing, self-protecting, self-optimizing, or self-configuring may be implemented as individual Touchpoint AMs monitoring the same managed resource.

Touchpoint AMs use policies to determine when and which corrective actions need to be performed on a managed resource [8]. Orchestrating AMs coordinate Touchpoint AMs. These managers can orchestrate within a discipline or across disciplines, i.e., coordinate on the same characteristics or on mixed autonomic characteristics within a system. Orchestrating managers ensure that touchpoint managers deliver autonomic capabilities to every part of the system where needed. The manual manager is the interface of the system, which allows for human computer interaction to take place. It is where human managers can specify the goals and policies that they would like performed within an autonomic system.

2.2 Current AC Research

Researchers all over the world are heavily investigating the field of autonomic computing. At Princeton University, work is being done in the areas of self-configuring and self-optimization in Impala [11]. Researchers at UC Berkeley are studying self-protection and self-healing in the OceanStore Project [10]. The University of Bologna, Telenor Communication AS, Technical University of Dresden, and Istituto Dalle Molle di Studi sull’Intelligenza Artificiale recently completed BISON, which is a shared project with the goal of creating self-managing systems that are based on various biological systems [3]. Despite the amount of research being done in the area of self-managing systems, little work is being done on designing frameworks to test them. The exploration of this field is at a stage where testing is critical to ensure that these newly developed systems are behaving as expected. By focusing on testing this an early stage of development, we hope to push towards a better understanding of these systems when they can be fully implemented.

2.2 Testing Approaches

There are two broad categories of testing – specification-based testing and implementation-based testing. Specification-based testing specifies the required testing in terms of identified features in the specification or requirements of the software.

[14]. This type of testing is not concerned with the actual implementation of the software. Implementation-based testing (or program-based testing) specifies the required testing in terms of the implementation (or program) under test [2, 14]. This type of testing makes sure that the program being run is exercised thoroughly. Implementation-based testing does not focus on any of the specifications of the software. Implementation based testing looks at factors such as branch coverage, statement coverage, line coverage, and path coverage.

Understanding the differences between specification-based and implementation-based allows the tester to identify and employ appropriate testing techniques. Although both approaches are considered to be equally important, the tester must be able to identify what kind of testing is required so that valuable time is not wasted performing too much or too little testing on different aspects of the software [2, 14].

2.3 Test Automation

Test automation requires careful planning during the software development process. It is estimated that test automation can take anywhere from 3 to 128 times longer to implement than manual testing [4]. Testing autonomic systems at runtime requires that testing be fully automated. We assume that the test automation process includes test execution, logging the test results, and evaluating the test results. To ensure the successful implementation of the test automation the following steps are required:

(1) The developer must specify functional requirements, high-level designs, and detail level designs to give to the test design engineer. Using this information the test design engineer creates the testing requirements, a test plan, and a set of test cases.

(2) A test environment must be set up so that the test cases can be executed during runtime of the autonomic system. Creation of the test environment includes accessing the test cases, test execution, result analysis, and verification of results. The commands for the test environment can be encoded in test scripts and executed while the autonomic system is running.

(3) A summary and evaluation of the test results should be generated after test execution. This includes determining what aspects of the software were tested and whether or not the tests were successful. Currently there are testing tools [5, 6] that can automatically execute and evaluate test cases, and calculate the level of test coverage acquired during testing.

3. AUTONOMIC CONTAINER

The main purpose of this work is to investigate and develop an approach for testing autonomic computing systems. Our method for attacking such a complex problem is to begin with a lightweight, simple application that has autonomic characteristics. We are testing such an application so that we can use it as a basis for how to test Autonomic Computing Systems. This application is the basis for our prototype and is referred to as an Autonomic Container.

An autonomic container can be defined as a data structure containing autonomic capabilities. Such a data structure is appropriate for preliminary investigation into testing autonomic computing systems. The small size and low complexity of the autonomic container gives a simplistic view of how autonomic systems are set up. Our conceptual autonomic container has self-protecting, self-healing, self-optimizing, self-configuring, and self-testing characteristics. However, the self-testing characteristic makes this application unique from other autonomic systems. Self-testing allows the autonomic container to be dynamically tested at runtime. Testing will generally take place when changes resulting from self-management are being invoked. This ensures that the overall functionality of the autonomic container is kept consistent. Such a set up makes it easier for problems in large systems to be identified and handled. As systems grow, errors and anomalies become harder to find due to the amount of complexity the system may contain. A self-test mechanism will deal with potential problems before these problems can cause some type of permanent damage to the system.

3.1 Architecture of an Autonomic Container

The architecture of an autonomic container consists of self-healing, self-optimizing, self-configuring, self-protecting and self-testing components as shown in Figure 3. We are using a stack for our autonomic data structure, and hence we include a stack class and the exceptions used by the stack. The stack has a

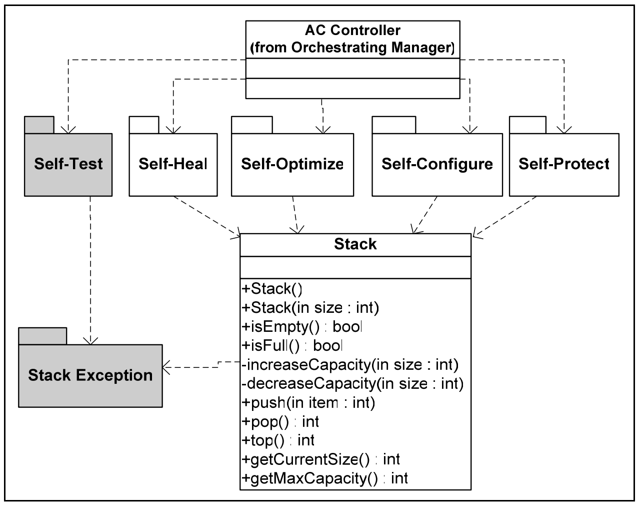

Figure 3: Architecture of Autonomic Container

public and private interface for use by an external application and the self-testing framework respectively. The interface of the stack in Figure 3 shows two of the methods used for self-configuring (`increaseCapacity` and `decreaseCapacity`) The AC_Controller class is part of the Orchestrating Manager, and is responsible for changes made to the autonomic container. It determines whether the system requires self-healing, self-configuring, self-optimizing, self-protecting, or self-testing. However, it is important to note that the self-testing functionality is an implicit capability of the Autonomic Manager, i.e., it is set up to be an assistant to the other characteristics rather than a new addition to an autonomic system.

4. Architecture of Self-Testing Autonomic Container

In this section we present the architecture of a testing framework for autonomic containers. Our architecture is based on a more general testing framework for AC systems, which is based on two approaches to test changes made to autonomic systems at runtime. The approaches include: (1) testing a copy of the changed component, and (2) testing the component during safe adaptation. In this section we describe the approach that tests a copy of a managed resource, while the actual managed resource is being used by an external application.

4.1 Overview of Architecture

Figure 4 shows a high-level architectural diagram for the testing framework for autonomic computing systems, using the approach that tests a copy of the changed component. It consists of six main components, which include the three components described in Section 2 (managed resource, touchpoint autonomic manager, and orchestrating manager), and three new components specific to the testing aspect of the framework (orchestrating test manager, touchpoint test manager and a copy of the managed resource).

In this paper we only describe the approach that tests a copy of the managed resource while the actual managed resource is being used by an application. The steps to test the copy of the managed resource (corresponding to Figure 4) are as follows:

1. The Touchpoint AM gathers information from the managed resource and recognizes that the resource requires a change, based on its self-management policy (i.e. self-healing, self-configuration, self-optimization, or self-protection).
2. A change request is then generated by the Touchpoint AM and this event is detected by its Orchestrating AM.
3. The Orchestrating AM sends a request to the Touchpoint AM to use the replication strategy (3a.) so that a copy of the managed resource may be tested. Simultaneously, a request is sent to the Orchestrating Test Manager to set up the validation process (3b.), which then creates the validation policy to be used during testing. This validation policy is then loaded into the Touchpoint Test Manager (3c.) to be used during the testing process. The policy includes the test coverage criteria, the location of the copy of the resource to be tested, and other constraints on the testing process (e.g., time allocated for testing).
4. The Touchpoint AM implements the change on the copy of the managed resource.
5. Once the change is fully implemented on the copy of the managed resource, a signal is sent to the Touchpoint Test Manager indicating that the copy is ready to be tested.
6. Test cases are then executed on the copy of the managed resource.
7. Test results are collected via a special interface in the Touchpoint Test Manager.
8. The Touchpoint Test Manager provides feedback to the Orchestrating Test Manager on whether the test performed passed or failed (8a.), and the adequacy of the test coverage (8b.).
9. The Orchestrating Test Manager then provides feedback to the Orchestrating AM on whether the change should be accepted (or rejected) due to the success (or failure) of validation.

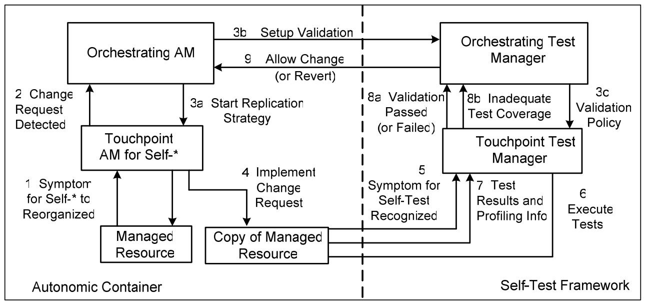
Orchestrating AM
3b Setup Validation
9 Allow Change (or Revert)
Orchestrating Test Manager
2 Change Request Detected
3a Start Replication Strategy
8a Validation Passed (or Failed)
8b Inadequate Test Coverage
3c Validation Policy
Touchpoint AM for Self-*
Touchpoint Test Manager
1 Symptom for Self-* to Reorganized
4 Implement Change Request
5 Symptom for Self-Test Recognized
7 Test Results and Profiling Info
6 Execute Tests
Managed Resource
Copy of Managed Resource
Autonomic Container
Self-Test Framework

Figure 4: High-Level Architecture of Testing Framework for Autonomic Container

5. PROTOTYPE

We developed a prototype of a self-testing autonomic container to validate our testing framework for AC systems. The autonomic container prototype contains all of the components required to provide the facilities of a self-managing data structure to an external application, and automatically perform runtime testing when self-management occurs. This section provides information on the framework setup environment and procedures used to develop the prototype, as well as the design and implementation details of the self-test manager. We also present the results of our experiment and discuss the uses and limitations of the prototype.

5.1 Framework Setup and Procedures

Our framework setup consisted of an external application that uses a data structure implemented within the autonomic container. This data structure is monitored by a self-configuring component, which interacts with the self-test component of the autonomic container. The overall strategy uses a system controller class that acts as an event coordinator. The system controller invokes the application that uses the data structure, and sets up the monitoring of the data structure's usage. When the system controller detects an attempt for self-configuration it invokes the self-test manager to perform runtime testing. Testing is performed using a copy of the data structure and the test results are analyzed to determine whether or not the change request should be accepted.

In addition, we utilized a mutation testing technique that implemented correct and incorrect versions of the stack class, entitled GoodStack and BadStack respectively. The purpose was to simulate a bad reconfiguration of the managed resource and observe whether or not our testing framework would detect that such a change should not be allowed.

Two third party testing tools were used in the prototype – JUnit [6] and Cobertura [5]. JUnit is a Java unit-testing framework from the xUnit family of testing frameworks. JUnit was used to execute test cases written for specific classes of the Autonomic Container. Cobertura is a Java code profiler that evaluates line and branch coverage of the unit testing activity and provides information on the code complexity. We used the report generator facility in Cobertura to automatically generate an HTML report containing the details of test coverage.

5.2 Design and Implementation

The prototype was developed in Java 5.0 using the Eclipse SDK and consisted of class implementations for the underlying data structure, a self-configuration component, testing software, and a program that manages the testing process. At the top level we have one main Autonomic Container package. This package is further decomposed into two sub-packages, SelfTestManager and StackExceptions, and includes a Stack class. The Stack class is the implementation of the underlying data structure to be used by the external application, and is therefore representative of the managed resource of the autonomic computing system.

The design of the SelfTestManager subpackage is shown in Figure 5. A JUnit test suite class (StackTest) was incorporated into this package along with a Cobertura coverage class (StackCover) class. The SelfTestManager also includes all of the files required for Cobertura to instrument the code and automatically calculate line and branch coverage. A batch file (TestRunner.bat) was created to implement the testing procedures in the SelfTestManager.

The StackExceptions package contains exception handling classes for erroroneous scenarios, and allows our simulation to detect events that warrant self-management. Since it is necessary to perform runtime testing, our Autonomic Container package also included a class for test execution (TestExec). The TestExec class allows other applications to be invoked at runtime, based on events detected from the actual operation of the Autonomic Container.

We developed a StackRunner class to simulate the environment needed to validate our newly created framework. StackRunner implements a random Boolean generator to determine whether or not to push or pop to an element onto the stack; while a random number generator is used to provide a value for the stack; and a looping mechanism to repeatedly run the application for a predetermined number of executions. The looping mechanism was set up to facilitate the observation, testing, and recording of changes to the system due to self-management and self-testing.

The first step of the testing procedures for the Autonomic Container involves setting Cobertura to instrument the Stack class. The JUnit tests from StackTest are then executed and the test results are written to a text file (test.log). Line and branch coverage data are then retrieved from the Cobertura datafile (cobertura.ser), and the StackCover class uses the values of the number of source lines of code and lines covered to compute the percentage of line and branch coverage achieved. This information is then appended to the test log and the Cobertura datafile is then deleted so that it will not affect data during the next self-test invocation. Upon completion of a self-testing session, all of the results of testing (i.e. number of unit test cases that passed/failed, along with line and branch coverage) are stored in the test log which can then be analyzed with respect to the validation policy for the Autonomic Container.

5.3 Results

The test suite for the autonomic container included 15 test cases developed using various test strategies such as: boundary, random and equivalence partitioning [14]. Parameterized test cases were used to validate properties of the data structure related to the capacity of the container. Testing criteria required 100% pass rate for all test cases, and a minimum of 75% for both branch coverage and statement coverage.

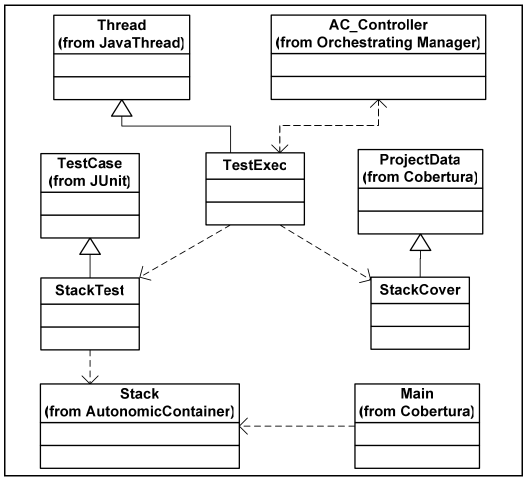

Figure 5: Design of Self-Test Manager

In the GoodStack implementation, the test case pass rate was 100% and the level of code coverage was 75% and 80% for branch and statement coverage respectively. For the BadStack, the test case pass rate was 86% (2 test failures). Hence, the GoodStack implementation passed the overall validation, while the BadStack implementation failed.

5.4 Discussion

Our prototype of the autonomic container provides evidence that supports the proposed approach to testing autonomic systems using copies of managed resources. The self-testing framework assumes that the autonomic container provides the following operations: (1) updating the copy of the managed resource e.g. increasing/decreasing the capacity of the stack, and (2) copying the contents of the container to the updated container after validation. The results for the GoodStack and BadStack simulations were favorable as validation failed for the incorrect stack, and hence would have prevented a potentially harmful change request from being implemented on the actual managed resource.

The current version of the prototype only implements the self-configuration and self-testing components of the autonomic container, and therefore must be further extended to observe more complex interactions within the SelfTestManager. Some core limitations of the current prototype include dynamic test case generation and code based changes. Analysis of the test results against the validation policy is also currently done manually.

Developing the prototype provided insight on the scope of the self-test manager with respect to operations that should be performed by the autonomic container and the self-testing framework. Even though we are only in the preliminary stages of this work, our results support the feasibility of our self-testing methodology for autonomic computing systems.

6. Related Work

Very little research has been done regarding the validation of autonomic systems. Validating autonomic systems is critical if we are to ensure that these systems function properly after self-management takes place. Developing a testing framework for autonomic systems is difficult for many reasons. Autonomic systems are extremely complex and it is very difficult to anticipate the environment in which autonomic elements operate [9].

Le Traon et al. [13] describe 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. Self-testable components can also be used as the building blocks for performing systematic integration and non-regression testing. The concept of self-testable components strongly supports the idea of self-testing in autonomic computing systems. In our approach we do not consider the notion of self-testable components. However, such components can be easily incorporated into our strategy thereby improving the overall approach.

Suliman et al. [12] describe an approach that uses the infrastructure of MORABIT (Mobile Resource Adaptive Built-in Test) to dynamically test components of a system. The approach allocates responsibilities of the testing process to the system components in order to minimize the effects of testing on the business logic execution. Our approach also dynamically tests a component (autonomic container), however we use the test manager to coordinate the activities of the testing process. The test manager also attempts to minimize the interruption of the autonomic container activities during testing.

7. Conclusions and Future Work

In this paper, we proposed a methodology that incorporates a dynamic self-test ability into an Autonomic Computing system. Our methodology validates changes in autonomic systems using copies of managed resources. The testing framework assumes that copies of managed resources are maintained in the autonomic system and are readily available for testing purposes. We validated our approach using a prototype that implements the self-configuration and self-testing components of an autonomic container.

Future work calls for investigation of more complex interactions within our testing framework by implementing other self-management characteristics such as self-healing, self-protection and self-optimization. We plan to extend the prototype to automatically perform analysis of test results based on a high-level validation policy. An XML parser will be added to the prototype to allow automatic extraction of data from the test reports generated by Cobertura, and compare this information with the validation policy stored in the knowledge base of the autonomic system.

We also intend to investigate the use of safe adaptation methods to allow self-testing to be performed directly on managed resources by placing the system in a partial operation state. Such a methodology would be used when managed resources either cannot be replicated or the cost of maintaining copies is too high.

8. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant IIS-0552555. The authors would like to thank the participants of the REU Summer 2006 program for their contributions to this work. In particular, the authors would like to thank our REU mentor Dr. Peter J. Clarke.

9. REFERENCES

[1] Automated Testing. First Steps in Test Automation, 2001. www.automatedtesting.com/FAQs_docs/firststeps.htm

[2] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Reading, Massachusetts, 2000.

[3] G. Di Caro, F. Ducatelle, and L.M. Gambardella. Project description: BISON: Biology-Inspired techniques for Self-Organization in dynamic Networks. *Zeitschrift Knstliche Intelligenz, Special Issue on Swarm Intelligence*, November 2005.

[4] H. Coetzee. Best Practices in Software Test Automation. *Test Focus Magazine* July/August 2005.

[5] Cobertura, July 2006. cobertura.sourceforge.net/

[6] E. Gamma and K. Beck. JUnit 3.8.1, Testing Resources, July 2006. www.junit.org.

[7] A. G. Ganek and T. A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 4(1), 2003 pages 5-18.

[8] IBM Corporation. An Architectural Blueprint for Autonomic Computing. IBM, June 2005.

[9] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer, 36(1),* pp. 41-50, January 2003.

[10] J. Kubiatowicz, B. Chun, S. Czerwinski.http://oceanstore. cs.berkeley.edu/info/overview.html

[11] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. Princeton University, 2003.

[12] D. Suliman, B Paech, L Borner, C Atkinson, D Brenner, M Merdes, R Malaka. The MORABIT Approach to Runtime Component Testing. *Second Int'l Workshop on Testing and Quality Assurance for Component-Based Systems (TQACBS 2006)*, pp. 171-176, IEEE Computer Society, 2006.

[13] Y. L. Traon, D. Deveaux, and J.-M. J´ez´equel. Selftestable components: From pragmatic tests to design for testability methodology. In *Technology of Object-Oriented Languages and Systems (TOOLS 99)*, pp 96–107. IEEE Computer Society, 1999.

[14] F H. Zhu, P.A.V. Hall, and J.H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4) pp. 366-427, December 1997.