Towards Self-Testing in Autonomic Computing Systems

Tariq M. King, Djuradj Babich, Jonatan Alava
and Peter J. Clarke
School of Comp. and Inform. Sciences
Florida International University
Miami, FL 33199, USA
email: {tking003, dbabi001, jalav001, clarkep}@cis.fiu.edu

Ronald Stevens
Dept. of Comp. and Inform. Sciences
Florida A & M University
Tallahassee, FL 32307, USA
email: rstevens@cis.famu.edu

Abstract

As researchers and members of the IT industry move towards a vision of computing systems that manage themselves, it is imperative to investigate ways to dynamically validate these systems to avoid the high cost of system failures. Although research continues to advance in many areas of autonomic computing, there is a lack of development in the area of testing these types of systems at runtime. Self-managing features in autonomic systems dynamically invoke changes to the structure and behavior of components that may already be operating in an unpredictable environment; further emphasizing the need for runtime testing. In this paper we propose a framework that dynamically validates changes in autonomic computing systems. Our framework extends the current structure of autonomic computing systems to include self-testing as an implicit characteristic. We validate our framework by creating a prototype of an autonomic container that incorporates the ability to self-test.

Keywords: Testing and Debugging, Autonomic Computing and Validation, Safe Adaptation.

1 Introduction

As technology-driven businesses continue to grow in size and complexity, there is a need to shift the burden of support tasks such as configuration, maintenance and fault management from people to technology [5]. IBM's autonomic computing paradigm is a movement towards self-managing systems that automatically configure, heal, protect, and optimize themselves [8, 10]. The paradigm has successfully attracted members from both the IT industry and the research community [9], and has led to the development of many research projects based on autonomic computing [12]. Throughout the paper we use the terms autonomic computing systems and autonomic systems interchangeably.

Although research continues to advance in many areas of autonomic computing, there is a lack of development in the area of testing autonomic computing systems at runtime. Dynamic changes and reconfigurations resulting from self-management are typically applied and accepted with little emphasis on validating them against the system requirements. Goals are set as high level policies that induce modifications to system components, and provide a way for system administrators to check if the system is operating within the bounds of some desired behavior. However, such an approach is insufficient to determine if the system behavior still conforms with the overall functional and nonfunctional requirements after such modifications have been implemented. The major contribution of this paper is the development of a methodology that supports automatic runtime validation of changes resulting from self-management in autonomic computing systems. Our methodology:

1. Automatically performs self-testing on change requests in autonomic computing systems at runtime, and evaluates test adequacy through the use of a validation policy file.
2. Extends the current architecture of autonomic computing systems by applying concepts of Orchestrating and Touchpoint Autonomic Managers [8] to testing activities.
3. Provides two strategies for validation based on system overhead cost and feasibility of use - safe adaptation with validation, and replication with validation.
4. Integrates testing into the current workflow of autonomic managers through communications to new autonomic managers designed for testing.

Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)
0-7695-2804-X/07 $20.00 © 2007 IEEE

This paper is organized as follows: the next section contains background information on autonomic computing, and briefly describes approaches to testing evolving and adaptive systems. Section 3 provides an overview of our testing methodology. Section 4 presents the architecture of the proposed autonomic testing framework. Section 5 provides details on the prototype used to validate our testing methodology. Section 6 presents related work, and in Section 7 we give concluding remarks and discuss future work.

2 Background

In this section we provide background information on Autonomic Computing, and outline IBM's architectural blueprint for building self-managing systems. Approaches to support the validation of evolving and adaptive systems are also discussed in this section.

2.1 Overview of 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 portrays a vision of computing systems [10] that manage themselves according to high-level objectives. Additional infrastructure is embedded within autonomic systems to automatically perform low-level decisions and actions, while administrators specify overall system behavior as high-level business-oriented policies. IBM identified the following core capabilities [10, 12] that support self-management in autonomic systems: (1) *self-configuration* – automatically configuring or re-configuring existing system components, and seamlessly integrating new components; (2) *self-optimization* – automatically tuning resources and balancing workloads to improve operational efficiency; (3) *self-healing* – proactively discovering, diagnosing and repairing problems resulting from failures in hardware or software; and (4) *self-protection* – proactively safeguarding the system against malicious attacks, and preventing damage from uncorrected cascading failures.

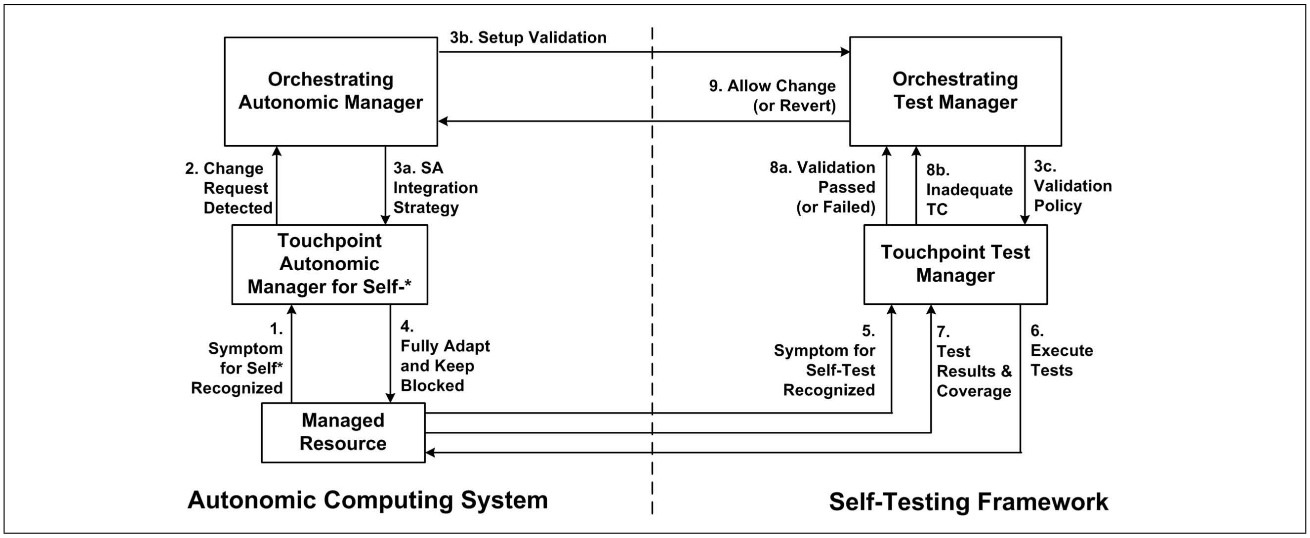
AC systems are characterized by intelligent closed loops of control that are typically implemented as the monitor, analyze, plan, and execute (MAPE) functions of autonomic managers (AMs). The monitor collects state information from the managed resource and correlates them into symptoms for analysis. If analysis determines that a change is needed, a change request is generated and a change plan is then formulated for execution on the managed resource.

2.2 Architectural Blueprint for AC

IBM's architectural blueprint for AC [8] defines a common layered approach for describing self-managing systems. The five building blocks for the blueprint are *touchpoints*, *autonomic managers*, *knowledge sources*, *a manual manager*, and an *enterprise service bus*. Touchpoints implement sensor and effector behavior for the manageability interfaces of *managed resources* [8, 10]. A managed resource is any type of hardware or software entity that can be managed. Autonomic Managers may be either Touchpoint AMs or Orchestrating AMs, and can be organized in an hierarchical fashion [10]. Touchpoint AMs work directly with managed resources through their touchpoints. Orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for particular resources. The portion of Figure 1 to the left of the dashed line depicts a typical layered setup for AMs in an autonomic computing system. An Orchestrating AM is shown managing a single Touchpoint AM, which in turn manages a single hardware or software resource. Knowledge sources are used to extend the built-in knowledge capabilities of AMs. At the topmost level is the manual manager (not shown in Figure 1), which implements a management console to facilitate the human administrator activity. An enterprise service bus connects the aforementioned building blocks by directing the interactions between them.

2.3 Testing Evolving & Adaptive Systems

The evolving nature of software means that systems typically require corrective, adaptive, or perfective maintenance [13] after initial deployment. Regression testing determines whether modifications to software have introduced new errors into previously tested code. This may involve re-running the entire test suite, or selecting a subset of the initial test suite for execution, i.e., *selective regression testing* [7]. Testing dynamically adaptive systems is extremely challenging because both the structure and behavior of the system may change during execution. Existing test cases may no longer be applicable due to changes in program structure, thereby requiring the generation of new test cases. *Safe adaptation* ensures that the integrity of system components is maintained during adaptation [17], and is comprised of the 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 adaption paths for performing adaptive actions



3b. Setup Validation
Orchestrating Autonomic Manager
9. Allow Change (or Revert)
Orchestrating Test Manager
2. Change Request Detected
3a. SA Integration Strategy
8a. Validation Passed (or Failed)
8b. Inadequate TC
3c. Validation Policy
Touchpoint Autonomic Manager for Self-*
Touchpoint Test Manager
1. Symptom for Self* Recognized
4. Fully Adapt and Keep Blocked
5. Symptom for Self-Test Recognized
7. Test Results & Coverage
6. Execute Tests
Managed Resource
Autonomic Computing System
Self-Testing Framework

Figure 1. A dynamic high-level test model for autonomic computing systems.

on system components. The actual adaptation is then performed during the realization phase in the following steps:

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 process' partial operation once all adaptive actions are complete.
4. Perform a local-post action to return the system to a fully-operational running state.

3 Overview of Testing Approach

Our approach incorporates testing activities into autonomic systems by providing validation services to autonomic managers (AMs) via test interfaces. Figure 1 shows our dynamic high-level test model for autonomic computing systems. The lefthand portion of Figure 1 shows a minimal autonomic computing system and the righthand portion shows our self-testing framework. The self-testing framework introduces test managers (TMs) that communicate with AMs to dynamically validate change requests. Our testing methodology can be divided into two general strategies which are based on system overhead cost and feasibility of use. Administrators can specify which strategy to use for individual resources depending on factors such as the nature of the resource (e.g., hardware or software); system configuration; and time and space requirements. In this section we introduce test managers and outline their duties, and provide detailed steps of our testing methodology based on the two strategies - *safe adaptation with validation*, and *replication with validation*. We describe the workflow of Figure 1 when using the aforementioned strategies in Subsections 3.2 and 3.3 respectively.

3.1 Test Managers

Test Managers (TMs) extend the concept of autonomic managers to testing activities. Like AMs, TMs may also be Orchestrating or Touchpoint [8]. Orchestrating TMs will coordinate high-level testing activities and manage Touchpoint TMs, while Touchpoint TMs perform low-level testing tasks on managed resources. We propose to integrate TMs within autonomic systems to dynamically validate the change requests generated by AMs. TMs will be responsible for:

- Performing regression testing to ensure that functional and nonfunctional requirements are still being met after the change;
- Generating test cases dynamically, and discarding test cases that are no longer applicable;
- Executing test cases and recording test results;
- Evaluating test results and test coverage with respect to a high-level validation policy;
- Maintaining a test repository to allow storage, organization and access to test cases, test logs, and validation policies.

To perform their duties, TMs will contain components that implement closed control loops that are consistent with the MAPE structure [8] of AMs.

3.2 Safe Adaptation with Validation

The *safe adaptation with validation* strategy validates changes resulting from self-management as part of a safe adaptation process [17], and occurs directly on the managed resource. This strategy can be used when it is too expensive, impractical, or impossible to duplicate managed resources. The overhead cost of this approach is defined by the amount of time the component under test remains blocked while waiting for validation to complete. The steps of this approach correspond to the workflow of Figure 1 (starting from the lefthand portion) as follows:

1. A Touchpoint AM gathers information from a managed resource and correlates it into a symptom that warrants self-managment.
2. A change request is generated by the Touchpoint AM and this event is detected via the sensor of an Orchestrating AM.
3. The Orchestrating AM initiates safe adapation (3a.) and concurrently requests that an Orchestrating TM sets up validation (3b.) of the resource. An appropriate validation policy (3c.) is then loaded into the knowledge component of a Touchpoint TM.
4. The Touchpoint AM proceeds with safe adaptation until the resource is fully-adapted (i.e., up to step 3 of the safe adaptation process outlined in Subsection 2.3) but keeps the resource blocked until validation is performed.
5. The Touchpoint TM detects that the resource is in a fully-adapted safe state, and is therefore ready to be tested.
6. After initial analysis and planning, the Touchpoint TM executes test cases on the managed resource.
7. Test results and test coverage are coalesced into a log file, which is analyzed by the Touchpoint TM with respect to the validation policy.
8. A message indicating whether validation passed or failed (8a.), or if there was inadequate test coverage (8b.), is then sent to the Orchestrating TM.
9. The Orchestrating AM is notified as to whether or not to accept or reject the change request. Based on the validation policy, the Orchestrating TM may either report to reject the change as a result of (8b.), or attempt to enhance the test suite and continue validation to acquire greater test coverage.

3.3 Replication with Validation

The *replication with validation* strategy requires the system to create and/or maintain copies of the managed resource for validation purposes. Change requests for managed resources are first implemented on copies and validated prior to execution on the actual managed resources. Using this strategy, step (3a.) of Figure 1 would notify the Touchpoint AM to use replication with validation instead of safe adaptation with validation. This would then redirect step (4.) to implement the change request on a readily available copy of the resource, and the Touchpoint TM would perform validation using the changed copy. Such a strategy is only feasible when managed resources can be replicated, and has the disadvantage of relatively high overhead costs with respect to the generation and maintainance of resource copies. However, the replication with validation strategy has the advantage that validation can occur without halting the regular operation of the system for the entire time required to test the component. In addition, validation does not have to occur on the same node as the autonomic system, and hence the computational and storage overhead can be shifted so that there is little or no affect on system performance.

4 Architecture of Touchpoint TMs

In this section we present the architecture of Touchpoint Test Managers and define the functions of their architectural components. We also describe the interactions between these components, and provide a high-level algorithm for validating change requests.

4.1 Description of Components

The architectural components of Touchpoint TMs apply MAPE functions in the context of testing activities, and are described as follows:

- *Test Monitor* - retrieves structural information about the change implemented in the resource under test, and collects test results and information on test coverage. Prior to validation, the test monitor continuously polls the resource to be tested, and recognizes when it is in a state suitable for validation to begin.
- *Test Analyzer* - performs analysis using the current test suite, previous structure of the resource, and the new structure of the resource to determine applicable regression test cases; and develops new test cases to enhance the current test suite. The

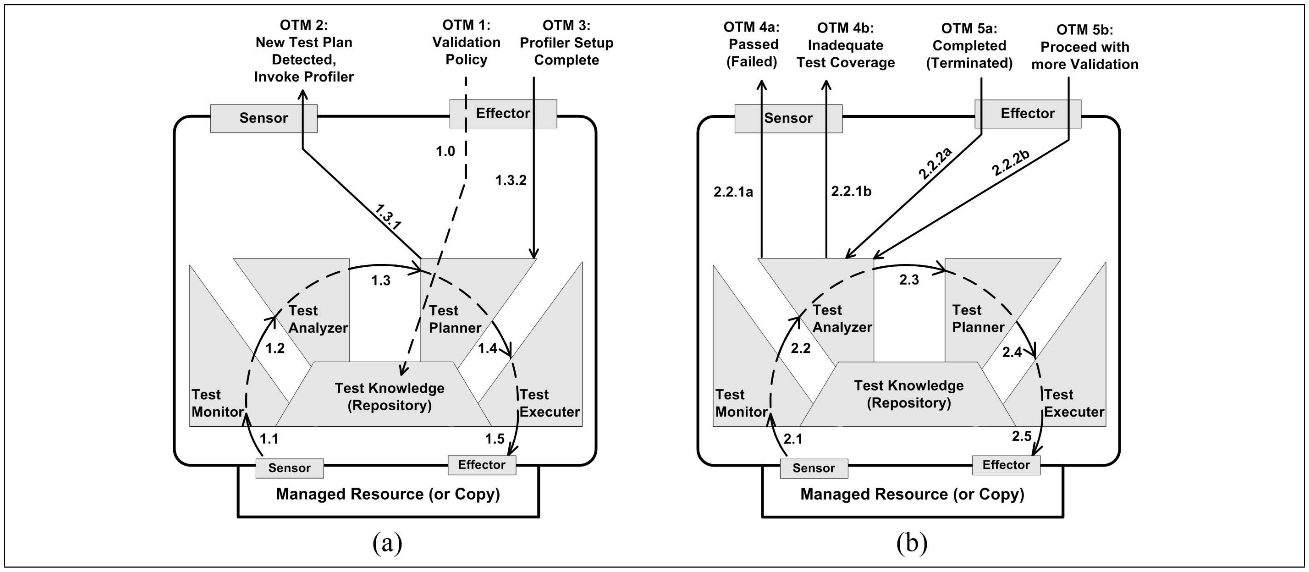


Figure 2. Component interactions through two closed control loops (a) and (b) in Touchpoint TMs.

test analyzer also evaluates the test results and test coverage provided in the test log.

- *Test Planner* - creates a test plan for the resource under test, which includes the test suite to be executed, a test schedule, and a post-test evaluation of the test log.
- *Test Executer* - applies test cases to the resource under test.
- *Test Knowledge* - stores validation policies that include the location of the resource to be tested, its previous structure, its current specification, a baseline test suite, and test coverage criteria. The test knowledge will also serve as a central repository for additional test cases, test results, and test histories.

4.2 Component Interactions

Like autonomic managers, TMs implement intelligent closed loops of control but for purposes of self-testing rather than self-management. These closed loops of control form the basis of the interactions between architectural components. Figure 2 shows two intelligent loops of control within Touchpoint TMs. The first loop traces arcs 1.1 through 1.5 in Figure 2(a), and the second loop traces arcs 2.1 through 2.5 in Figure 2(b). The operation of these loops may be externally monitored and intercepted 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. Figure 3 shows a high-level algorithm which corresponds to how the closed loops and external orchestrating actions work together to validate changes to managed resources. The discussion for the rest of this subsection provides details for the relationship between Figures 2 and 3.

```
Begin ValidateChangeRequest
    1.0 Load Validation Policy
   // First Intelligent Closed Control Loop
    1.1 Retrieve new structure
    1.2 Perform test suite analysis
    1.3 Request to create test plan
      // External Monitored Event
        1.3.1 TestPlanCreated
      // Corresponding Actuated Event
        1.3.2 FinalizeTestPlan
    1.4 Execute test plan
    1.5 Run test cases
   // Second Intelligent Closed Control Loop
     2.1 Retrieve test results
     // External Monitored Events
       2.2.1a TestsPassed or TestsFailed
       2.2.1b InadequateCoverage
     // Corresponding Actuated Events
       2.2.2a StopTesting (goto End)
       2.2.2b ReanalyzeTestSuite (goto 1.2)
End ValidateChangeRequest
```

Figure 3. Change validation algorithm.

An Orchestrating TM first sends a validation policy for the resource to a Touchpoint TM, which loads it into the test knowledge component (1.0). The Touchpoint TM will automatically invoke the test monitor to retrieve the new structure of the resource (1.1) when it senses that the resource is ready to be validated. The test monitor then sends the information about the new structure to the test analyzer (1.2), which uses it in

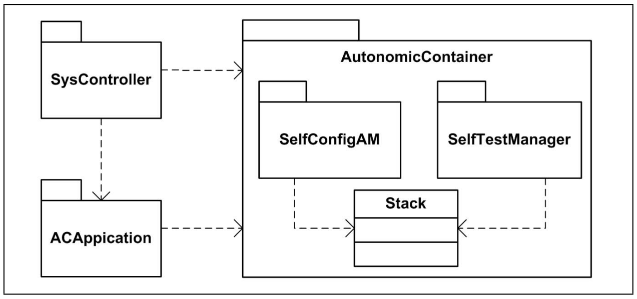


Figure 4. Top-level design of prototype.

conjuction with the previous structure and a baseline test suite to prepare a new test suite. The test analyzer notifies the test planner that a new test suite is ready and requests that a test plan be created (1.3). At this point, the Orchestrating TM detects that a TestPlanCreated event (1.3.1) has occurred and intercepts the closed loop to set up an external code profiling tool. Once the code profiler has been setup to instrument the managed resource for test coverage, the test plan is finalized and control is returned to the closed loop (1.3.2). The test plan is then passed to the test executer component (1.4) which then starts running test cases on the managed resource (1.5), and initiates the second closed control loop.

The second intelligent control loop commences with the retrieval of the test results and test coverage information (2.1) from the managed resource. The test monitor correlates this information into a test log and passes it to the test analyzer (2.2). The test analyzer evaluates the test log against the validation policy and determines whether validation passed or failed (2.2.1a), or if the test suite was inadequate (2.2.1b) with respect to the test coverage criteria. The results of the evaluation are sent to the Orchestrating TM, which would either end the validation process (2.2.2a), or continue testing after re-analyzing the current test suite (2.2.2b). If a decision is made to re-analyze the test suite, the behavior of the second loop from 2.3 to 2.5 in Figure 2(b) is the same as the first loop from 1.3 to 1.5 in Figure 2(a).

5 Prototype

To validate our approach we developed a prototype of a Java application that has autonomic capabilities. Our prototype implements a data structure that has the ability to self-configure at runtime, which we refer to as an *autonomic container*. The managed element of the autonomic container was implemented as a stack, and an external application that makes random requests to push (pop) data elements to (from) the stack was developed. When the stack is at an average of 80 percent of its full capacity, the AM for self-configuration attempts to reconfigure the stack by increasing its capacity. This reconfiguration is intercepted and a signal is sent the self-testing component to validate the change request on a copy of the stack. Two tools were used to support testing: JUnit [6] and Cobertura [4]. In this section we provide the design of the prototype, describe the setup of the test environment, and discuss uses and limitations of the prototype.

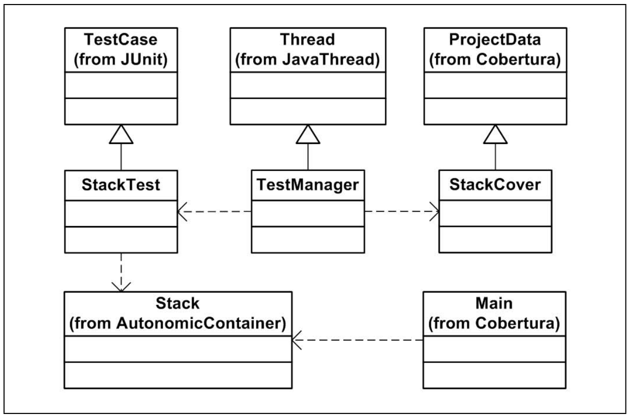


Figure 5. Design of self-test component.

5.1 Design of Prototype

At the top level, the prototype contains three main packages: ACApplication, AutonomicContainer, and SysController, as shown in Figure 4. ACApplication contains the classes that make up the external application that uses the AutonomicContainer. The application has two basic modes of operation - *Random* and *AlwaysPush*. Random mode determines whether to push or pop data elements by generating randomized boolean values, whereas the AlwaysPush mode continuously pushes data elements onto the stack. In both modes, the values of the data elements for push operations are random numbers. The package AutonomicContainer incorporates our self-testing framework. It contains the nested packages SelfConfigAM and SelfTestManager that implement intelligent control loops for self-configuration and self-testing respectively; as well as the Stack class and its manageability interface.

The design of the self-test component SelfTestManager is shown in Figure 5. The class TestManager implements functionality for the test monitor, test analyzer, test planner, test executer, and test knowledge. Test cases for the stack were developed using the JUnit superclass TestCase and stored in the class StackTest. Branch and line coverage were acquired by extending the class ProjectData from Cobertura, and having its Main class instrument the Stack class of the autonomic container for code coverage.

The validation policy for the SelfTestManager is stored in XML format and contains the test adequacy criteria and previous structure of the stack. Test results and test coverage are also stored in an XML file, which is supplied to the class TestManager for evaluation against the validation policy. The SysController package invokes the parallel execution of the ACApplication and AutonomicContainer through the use of the Java threads.

5.2 Setup Environment

We developed the prototype in Java 5.0 using the Eclipse 3.1 SDK, along with the necessary plugins and packages for the testing support tools JUnit and Cobertura. JUnit [6] is a unit testing framework for Java from the xUnit family of testing frameworks. JUnit v3.8.1 was used to develop and automatically execute unit test cases on the autonomic container. Cobertura 1.8 [4], a Java code profiler, was used to automatically evaluate line and branch coverage as the test cases were being applied to the system. Cobertura was also used to generate reports on test coverage in XML format.

5.3 Discussion

The main objective of developing the prototype was to validate the high-level test model for autonomic computing systems shown in Figure 1. Although the prototype implements a minimal autonomic container, it provides evidence to support that the replication with validation strategy is feasible. To simulate the behavior of the test model in Figure 1, the autonomic system and self-testing framework were implemented as separate threads communicating through shared variables and synchronized method calls. Our self-testing framework assumes that the autonomic container provides the following operations: (1) updating the copy of the managed resource (e.g., increasing the stack capacity), (2) copying the contents of the container to the updated container after validation, and (3) providing a mechanism for safe adaptation [16].

The high-level validation policy for the prototype required 100% pass rate for all test cases executed, and at least 75% branch and statement coverage. The initial test suite for the autonomic container included 15 test cases developed using a combination of test strategies. These test strategies included: boundary, random, and equivalence partitioning [18]. Test cases to validate properties related to the capacity of the container were parameterized. To evaluate the prototype for the testing framework, we utilized a mutation testing technique to simulate two change request scenarios: (1) a request for correct re-configuration of the stack, and (2) a request for incorrect re-configuration of the stack. The latter was achieved by altering the method that re-configures the stack capacity to cause a decrease in capacity rather than an increase. In the first scenario, all of the test cases passed and the code coverage criteria were met (75% branch, 80% statement). The second scenario produced two test case failures, and therefore code coverage was omitted. These results were favorable as validation failed for the incorrect re-configuration scenario, and hence would have prevented a potentially harmful change request from being implemented on the stack.

Building the prototype provided us with insight on the scope of the self-test manager in terms of the operations that should be performed by the autonomic system and the self-testing framework. Dynamic test case generation and code-based changes remain core limitations of the prototype. Currently the prototype only implements replication with validation.

6 Related Work

Although there is clearly a need for autonomic systems to perform self-testing [10, 12], few researchers have addressed this issue. To the best of our knowledge this is the first work that incorporates self-testing as an integral part of autonomic systems.

Several researchers have investigated the notion of self-testing software [1, 2, 11, 14, 15]. The work by Le Traon et al. [14] is most 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. 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.

Denaro et al. [3] present an approach that automatically synthesizes assertions from the observed behavior of the application aimed at achieving 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, which can be used to test components after an autonomic change has been implemented.

Zhang et al. [17, 16] present an approach that formalizes the behavior of adaptive programs using state machines and Petri nets, respectively. The approach separates the adaptation behavior from the non-adaptive behavior, making the models easier to specify and verify using automated techniques. The contributions of the work by Zhang and Chen [16] that can be applied to our work include: (1) specification of global invariants of the properties of adaptive programs regardless of the adaptations and (2) creation of state-based models that aid in the generation of rapid prototypes. Using the specifications for global invariants and state-based models, test cases can be dynamically generated to test an adapted program at runtime.

7 Conclusions and Future Work

In this paper we proposed a framework that dynamically validates changes in autonomic computing systems. Our framework extends the current structure of autonomic computing systems to include self-testing as an implicit characteristic. We presented two strategies to support testing autonomic computing systems at runtime: safe adaptation with validation, and replication with validation. To show the feasibility of our testing framework, we developed a prototype that uses a minimal autonomic computing system. Our future work involves extending the capabilities of the prototype to include operations for self-healing, self-optimizing, and self-protecting. We also plan to finish implementing the safe adaptation with validation strategy in the prototype.

8. Acknowledgements

This work was supported in part by the National Science Foundation under grant IIS-0552555. The authors would like to thank the participants of the FIU REU Summer 2006 program and Dr. Masoud Sadjadi for their contributions to this work.

References

[1] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 45(6):549595, 1993.

[2] G. Denaro, L. Mariani, and M. Pezzè. Self-test components for highly reconfigurable systems. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.

[3] G. Denaro, L. Mariani, M. Pezzè, and D. Tosi. Adaptive runtime verification for autonomic communication infrastructures. In *First International IEEE WoWMoM Workshop on Autonomic Communications and Computing (ACC'05)*, pages 553–557, 2005.

[4] M. Doliner, G. Lukasik, and J. Thomerson. Cobertura 1.8, 2002. `http://cobertura.sourceforge.net/` (August 2006).

[5] Enterprise Managment Associates. Practical autonomic computing: Roadmap to self managing technology. Technical report, IBM, Boulder, CO, Jan. 2006.

[6] E. Gamma and K. Beck. JUnit 3.8.1, 2005. `http://www.junit.org/index.htm` (August 2006).

[7] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[8] IBM Autonomic Computing Architecture Team. An architectural blueprint for autonomic computing. Technical report, IBM, Hawthorne, NY, June 2006.

[9] IBM Corporation. IBM Research, 2002. `http://www.research.ibm.com/autonomic/` (August 2006).

[10] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, January 2003.

[11] R. Kumar and D. Sivakumar. On self-testing without the generator bottleneck. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 248–262, London, UK, 1995. Springer-Verlag.

[12] H. A. Muller, L. O'Brien, M. Klein, and B. Wood. Autonomic computing. Technical report, Carnegie Mellon Univeristy and Software Engineering Institute, April 2006.

[13] I. Sommerville. *Software Engineering: Seventh Edition.* Addison-Wesley, Essex, England, 2004.

[14] Y. L. Traon, D. Deveaux, and J.-M. Jézéquel. Self-testable components: From pragmatic tests to design-for-testability methodology. In *Technology of Object-Oriented Languages and Systems (TOOLS 99)*, pages 96–107. IEEE Computer Society, 1999.

[15] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

[16] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM Press.

[17] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley. Enabling safe dynamic component-based software adaptation. In *WADS*, pages 194–211, 2004.

[18] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit testing coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.