An Integrated Self-Testing Framework for Autonomic Computing Systems

Tariq M. King, Alain E. Ramirez, Rodolfo Cruz, and Peter J. Clarke
School of Computing and Information Sciences, Florida International University, Miami, USA
Email: {tking003, aeste010, rcruz002, clarkep}@cis.fiu.edu

Abstract— **As the technologies of autonomic computing become more prevalent, it is essential to develop methodologies for testing their dynamic self-management operations. Self-management features in autonomic systems induce structural and behavioral changes to the system during its execution, which need to be validated to avoid costly system failures. The high level of automation in autonomic systems also means that human errors such as incorrect goal specification could yield potentially disastrous effects on the components being managed; further emphasizing the need for runtime testing. In this paper we propose a self-testing framework for autonomic computing systems to dynamically validate change requests. Our framework extends the current architecture of autonomic systems to include self-testing as an implicit characteristic, regardless of the self-management features being implemented. We validate our framework by creating a prototype of an autonomic system that incorporates the ability to self-test.**

Index Terms— **autonomic computing, testing, validation.**

I. Introduction

Continuous technological advances have led to an unprecedented growth in the size and complexity of computing systems within the last two decades [1]. The human and economic implications of this growth are compounded by the need for large-scale system integration, in order to facilitate collaboration among multiple business enterprises. Managing complex, large-scale, interconnected computing systems requires a team of highly skilled IT professionals to provide ongoing technical support services. Therefore, as technology-driven businesses continue to expand, they incur increasingly high costs to maintain their computing infrastructure. In addition, some computing infrastructures are now so complex that it has become almost impossible to manage them manually. Major industrial players such as IBM have therefore recognized the need to shift the burden of support tasks such as configuration, maintenance and fault management from people to technology [2].

Autonomic computing is a movement towards self-management computing technology that aims to reduce the difficulties faced by administrators when managing complex systems. The paradigm emphasizes computing systems that automatically configure, optimize, heal, and protect themselves [3], [4], in accordance with administrator objectives. IBM has successfully attracted members from both the IT industry and the academic community to the area of autonomic computing through various manifestos, research papers, and technical reports [1]–[5]. The initiative continues to stimulate great interest in the scientific community, and has led to the development of numerous projects based on autonomic computing [6].

Although research is advancing in many areas of autonomic computing, there is a lack of development in the area of testing autonomic systems at runtime. Runtime testing is necessary in autonomic systems because self-management features induce structural and behavioral changes to the system during its execution. Furthermore, the high level of automation in autonomic systems means that incorrect goal specification could yield potentially disastrous, and even irreversible effects on the components being managed. This research seeks to address two fundamental questions associated with runtime changes in autonomic computing systems. These questions are: (1) After a change, how can we be sure that new errors have not been introduced into previously tested components? and (2) If a change introduces a new component, how can we be sure that the component will actually behave as intended?

In this paper we propose a self-testing framework for autonomic computing systems that dynamically validates change requests through regression testing, and the execution of additional tests which exercise the behavior of newly added components. Our framework integrates testing into the current workflow of autonomic managers through communications to new autonomic managers designed for testing. We apply concepts of autonomic managers, knowledge sources, and manual management facilities [3] to testing activities for autonomic systems. Our testing methodology is based on two general validation strategies – *safe adaptation with validation*, and *replication with validation*.

We extend the previous work by King et. al [7] to provide a more comprehensive self-testing framework with additional components that enhance test coordination and facilitate manual test management. We describe the duties of these new components and present examples of their use within the integrated framework. In addition, we provide descriptions of the interactions between these new components and the existing framework components, and

This paper is based on "Towards Self-Testing in Autonomic Computing Systems," by T.M. King, D. Babich, J. Alava, R. Stevens, and P.J. Clarke, which appeared in the Proceedings of the 8th IEEE International Symposium on Autonomous Decentralized Systems (ISADS), Sedona, USA, March 2007. © 2007 IEEE.

This work was supported in part by the National Science Foundation under grants IIS-0552555 and HRD-0317692.

discuss the role played by the human administrator when managing the testing process. We also present an extended implementation of a prototype developed to show the feasibility of our testing methodology.

This paper is organized as follows: the next section describes the autonomic computing paradigm and related software testing approaches. Section III presents the architectural model of the self-testing framework, and outlines the steps of our testing methodology. Section IV discusses the component interactions of the test managers of the framework. Section V presents the implementation details of the prototype. Section VI presents the related work, and in Section VII we give concluding remarks.

II. Autonomic Computing and Testing

In this section we provide a brief overview of the autonomic computing paradigm, and describe IBM's architectural blueprint for building autonomic systems. We also summarize the main approaches to software testing, and outline the major steps and challenges of test automation. Techniques to support runtime testing of evolving and adaptive systems are also discussed in this section.

A. 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 was launched in October 2001 and portrayed a vision of computing systems [4] 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, which regulates vital bodily functions such as homeostasis without the need for conscious human involvement [1]. AC extends this biological concept to computing systems by embedding additional infrastructure within the system to carry out low-level decisions and tasks, while administrators specify overall system behavior as high-level policies. The core capabilities that support self-management in autonomic computing systems [4], [6] include:

Self-configuration. Automatically configuring or reconfiguring existing system components, and seamlessly integrating new components.

Self-optimization. Automatically tuning resources and balancing workloads to improve operational efficiency.

Self-healing. Proactively discovering, diagnosing and repairing problems resulting from failures in hardware or software.

Self-protection. Proactively safeguarding the system against malicious attacks, and preventing damage from uncorrected cascading failures.

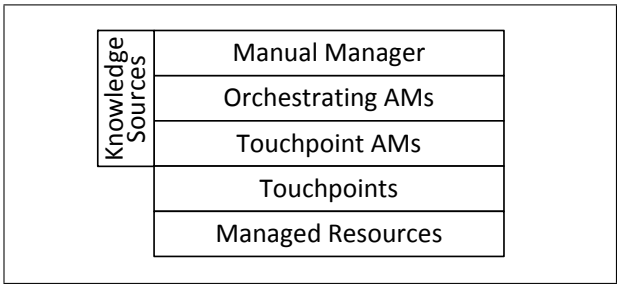

Figure 1. Layered architecture of Autonomic Computing.

In order to accomplish the aforementioned self-management tasks, autonomic systems must be able to observe their own structure and behavior (*introspection*); adapt or modify their own structure and behavior (*intercession*) [8]; and be aware of their environmental and operational contexts. Self-awareness is therefore an inherent characteristic of AC systems regardless of the specific autonomic capabilities being implemented.

B. Architecture for Autonomic Computing

The architectural blueprint for AC [3] defines a common layered approach for developing self-managing systems as shown in Figure 1. The horizontal layers include *managed resources*, *touchpoints*, *touchpoint autonomic managers*, *orchestrating autonomic managers*, and *a manual manager*. A vertical layer of *knowledge sources* (top-left of Figure 1) spans the top three horizontal layers to facilitate the exchange, retrieval, and archival of management information among these layers.

The managed resource layer consists of the hardware or software entities for which self-management services are being provided. Directly above the managed resources are manageability interfaces called touchpoints. The touchpoints implement the sensor and effector behaviors [3], [4] necessary to automate low-level management tasks. Sensors provide introspection mechanisms for gathering details on the current structure or behavior of managed resources, while effectors provide intercession mechanisms to facilitate structural or behavioral modifications.

A higher level of management is provided by autonomic managers (AMs). There are two categories of AMs, namely Touchpoint AMs and Orchestrating AMs [3]. Touchpoint AMs work directly with managed resources through their touchpoint interfaces. Orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for individual resources. The topmost layer is an implementation of a management console, called the manual manager, which facilitates the human administrator activity. The vertical layer of knowledge sources implements registries or repositories that can be used to extend the capabilities of AMs, and may be directly accessed via the manual manager.

Autonomic systems are characterized by intelligent closed loops of control, in which sensed changes to managed resources result in the invocation of the appropriate set of actions required to maintain some desired system state. In autonomic systems, these closed control loops are

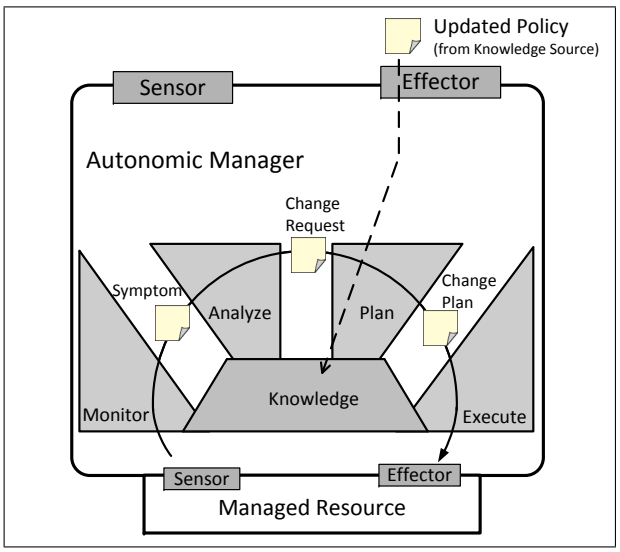

Figure 2. MAPE structure of autonomic managers.

typically implemented as the *monitor*, *analyze*, *plan*, and *execute* (MAPE) functions of AMs as shown in Figure 2. 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. AMs contain a knowledge component which allows access to data shared by the MAPE functions. This built-in knowledge component generally contains information relating to self-management policies, and interacts with the knowledge sources shown in Figure 1 to provide an extensible self-management framework. For example, as shown in Figure 2, extensibility in autonomic systems can be achieved by loading a new policy from a knowledge source into the built-in knowledge of the AM to update its capabilities.

C. Software Testing Approaches

The two main categories of software testing are *specification-based* and *program-based* testing. Specification-based testing is based soley on the definition of what the program under test is supposed to do and employs no internal information about the structure of the program [9]. The specification drives the testing process by providing a means of checking the correctness of output (*test oracles*); selecting test data; and measuring test adequacy. On the other hand, program-based testing defines testing requirements strictly in terms of whether or not the program under test has been thoroughly exercised [9]. This category of testing is usually associated with some form of test adequacy criteria that is based on coverage of particular elements of the program, e.g., *all statements*, *all branches*, or *all paths* [10].

Understanding the differences between specification-based and program-based testing allows the software tester to select appropriate testing techniques under given circumstances. 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 and resources are not wasted performing too much or too little testing on different aspects of the software.

The notion of test data adequacy is central to any testing methodology 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 [10]:

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 of this set 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 [10]. However, as a stopping rule, a test data adequacy criterion C is a special case of measurements with the range $\{0, 1\}$, i.e., $false$ or $true$.

The effectiveness of a test set with respect to its ability to reveal faults can be assessed using a technique known as *mutation testing* [9], [10]. 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 [10]. 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.

D. Test Automation

Test automation calls for careful planning during the software development process, and is impossible without tool support. Automating the testing process involves designing test cases; creating test scripts; and setting up a test harness for automatically executing tests, logging results, and evaluating test logs [11]. 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. There are several categories of tools that support test automation. These include test design tools, dynamic analysis tools, GUI test drivers, capture/playback mechanisms, and test evaluation tools [11].

Most testing tool vendors boast that cost savings and reduction in defects are some of the benefits to be gained from using their products [12]. However, test automation still presents several challenges to the test engineers. For example, the capture/playback method of

automation which is prevalent in many vendor packages does not adequately address the needs of complex, multi-environment systems [12]; thereby leaving test engineers to manually implement supporting programs to achieve the desired level of automation. In addition, some testing tools only support a specific granularity of testing such as unit, integration, or system testing. Successful test automation therefore requires the selection of an appropriate combination of testing tools, and a skilled test automation team to develop support programs and perform traditional testing tasks.

E. Testing Evolving and Adaptive Systems

As previously mentioned in Section II-A, autonomic computing addresses the software complexity and evolution problems through the use of a highly adaptive self-management infrastructure. Software evolution is inevitable due to the emergence of new requirements, changes in the operating environment, and the need to fix newly discovered errors. As a result, software systems typically require perfective, adaptive, or corrective maintenance [13] after initial deployment. Regression testing determines whether modifications to software due to maintenance have introduced new errors into previously tested code [9]. This may involve re-running the entire test suite (*retest-all*), or selecting a subset of the initial test suite for execution (*selective regression testing*) [14]. Techniques for regression test selection include dataflow, random, safe and test minimization [14].

Testing dynamically adaptive systems is extremely challenging because both the structure and behavior of the system may change during its execution. Existing test cases may no longer be applicable due to changes in program structure, thereby requiring the generation of new test cases. Modern programming languages such as Java and C# support introspection of the structure of an object at runtime, and hence provide mechanisms for extracting useful information for dynamic test case generation.

A disciplined approach to adaptation can also be used to support runtime testing of adaptive systems. *Safe adaptation* ensures that the integrity of system components is maintained during adaptation [8]. The safe adaptation process 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 on system components. The actual adaptation is then performed during the realization phase in the following steps:

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

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

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

Step 4. Perform a local-post action to return the system to a fully-operational running state.

The safe adaptation process will not violate any dependency relationships or interrupt critical communication between components, even in the presence of failures [8]. 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 before a rollback or the adaptation process runs to completion.

III. Autonomic Self-Testing Framework

The adaptive and evolving nature of AC systems requires that testing be an integral part of these systems. We affirm that, similar to the inherent self-awareness characteristic of AC systems discussed in Section II-A, self-testing is an implicit characteristic of autonomic systems regardless of the specific self-management features being implemented. Self-testing in autonomic systems is necessary for dynamically: (1) performing regression testing after changes have been implemented to ensure that new errors have not been introduced into previously tested components; and (2) validating the actual behavior of newly added or adapted components prior to their use in the system.

We propose an integrated self-testing framework for autonomic computing systems to perform the aforementioned tasks, thereby addressing the research questions posed in Section I. Our self-testing framework augments the dynamic high-level test model for autonomic computing systems in [7] with components that enhance test coordination, and provide detailed descriptions of their interactions. In order to be consistent with the grand vision of autonomic computing, our self-testing framework fully automates testing activities wherever possible, and seeks to minimize the administrator's effort in those testing activities which require manual intervention.

As shown in Figure 3, our approach incorporates testing activities into autonomic computing systems by providing validation services to autonomic managers (AMs) and the manual manager implementation via test interfaces. The lefthand portion of Figure 3 shows an autonomic computing system and the righthand portion shows the self-testing framework. The self-testing framework consists of *test managers*, *test knowledge sources* and *auxiliary test services* that collaborate to provide validation services for the autonomic system.

Our testing methodology is based on two general strategies – *safe adaptation with validation* and *replication with validation*, which differ with respect to system overhead cost and feasibility of use. Depending on the strategy used, the autonomic system may be required to maintain

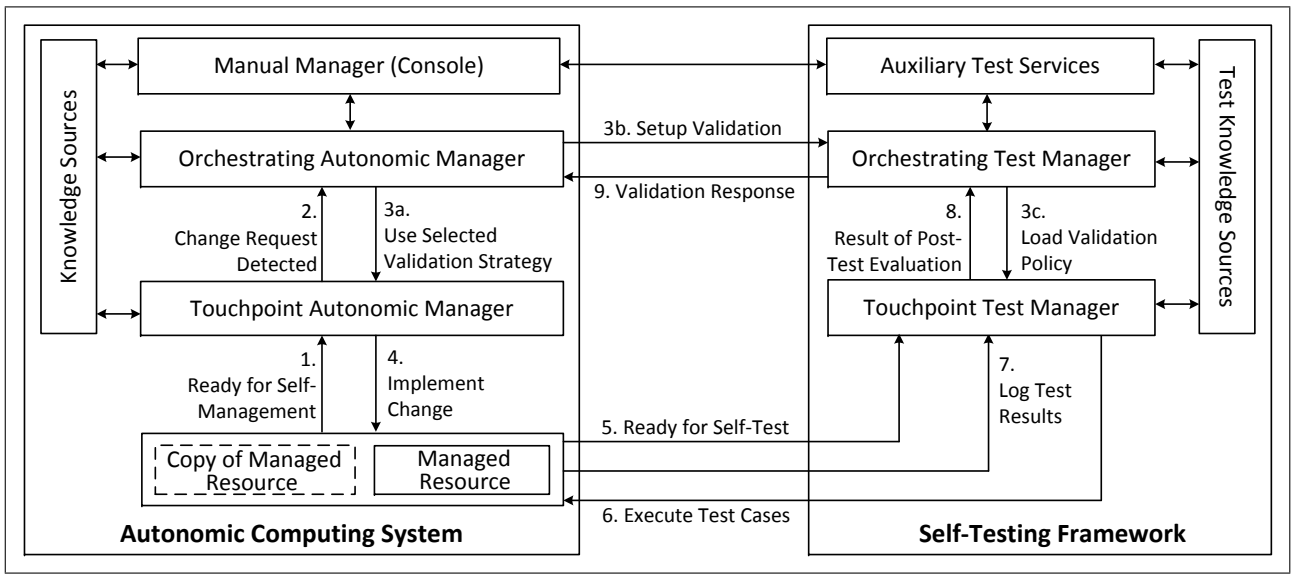

Figure 3. High-level architectural model for an integrated self-testing framework for autonomic computing systems.

copies of managed resources for testing purposes as illustrated by the dashed box (bottom-left of Figure 3). The self-testing framework also assumes that the autonomic system provides mechanisms for implementing change requests on the copy of the managed resource; transferring the contents of the copy to the actual managed resource after validation; and performing safe adaptation [15].

In this section we present the proposed autonomic self-testing framework and describe its major components. Subsections III-B and III-C provide the detailed steps of our testing methodology, in the context of the two validation strategies, through descriptions of the workflow of Figure 3. We also provide the rationale and guidelines for applying each validation strategy, and discuss how we plan to address some of the key challenges of testing autonomic systems at runtime.

A. Test Managers

Test managers (TMs) extend the concept of autonomic managers to testing activities. Like autonomic managers, TMs may also be Orchestrating or Touchpoint. Orchestrating TMs coordinate high-level testing activities and manage Touchpoint TMs, while Touchpoint TMs perform low-level testing tasks on managed resources. During system execution, Orchestrating AMs will notify Orchestrating TMs that some change request requires validation. Orchestrating TMs will then coordinate the testing activities necessary to validate that change request using the managed resource or a copy. TMs will be responsible for:

Test coordination. Directing the end-to-end validation process by interfacing with AMs; checking for new validation policies in test knowledge sources; coordinating the tasks of Touchpoint TMs; and invoking auxiliary test services.

Test planning and execution. Scheduling and performing regression testing and, if necessary, executing newly generated test cases to validate change requests for managed resources.

Test suite management. Generating new test cases dynamically, and discarding test cases that may no longer be applicable due to changes in system structure.

Pre- and post-test setup. Setting up the test environment, and preparing a test log to be used during the post-test evaluation.

Post-test evaluation. Analyzing and evaluating test results and test coverage provided in the test log against the high-level validation policy.

Storage of test artifacts. Maintaining a repository to store test cases, test logs, test histories, and validation policies.

To perform the aforementioned duties, TMs will contain components that are consistent with the MAPE structure [3] of autonomic managers. A *test monitor* will be responsible for polling the managed resource, and/or various components within the self-testing framework, to collect any information relevant to the testing process. A *test analyzer* will then determine whether or not some testing-related activity needs to be performed such as setting up the test environment, generating new test cases, or conducting a post-test evaluation. A *test planner* will then generate a plan for the testing activity, and a *test executer* component will perform the required testing tasks. A *test knowledge* component will serve as a central repository for test artifacts, and coordinate the interactions between the aforementioned components. Detailed descriptions of the component interactions within Orchestrating TMs and Touchpoint TMs are provided in Subsections IV-A and IV-B, respectively.

B. Safe Adaptation with Validation

The safe adaptation with validation strategy validates changes resulting from self-management as part of a safe adaptation process [8], and occurs directly on the

managed resource during system execution. The steps of this approach as corresponds to the workflow of Figure 3, starting from its bottom lefthand portion, are as follows:

---

1. Ready for self-management. A Touchpoint AM gathers information from a managed resource and correlates it into a symptom that warrants 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. Safe adaptation with validation. The Orchestrating AM initiates safe adapation 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 II-E) but keeps the resource blocked until validation is performed.

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 test 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 or failed, or if there was inadequate 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.

The safe adaptation with validation strategy should be used if it is too expensive, impractical, or impossible to duplicate managed resources. Recall that in autonomic systems, managed resources may be either hardware or software entities. From a business perspective, it may not be economically viable to maintain duplicates of some hardware or software components soley for testing purposes. In addition, instantiating and executing copies of software components may affect system performance negatively and hence may not be practically feasible. Safe adaptation with validation addresses these issues by allowing testing to be performed directly on managed resources. However, this approach has the disadvantage that some application services may have to be temporarily suspended while the validation process completes.

C. Replication with Validation

The replication with validation strategy requires the system to create and/or maintain copies of the managed resource for validation purposes (recall dashed box at bottom-left of Figure 3). Change requests for managed resources are first implemented on copies and validated prior to execution on the actual managed resources. Using this strategy, the steps of our testing methodology as corresponds to Figure 3 differ as follows:

---

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.

The replication with validation strategy should be used when managed resources can be easily replicated. It has the advantage that the regular service of the system does not have to be suspended for the entire time required to perform testing. Furthermore, if the component under test is hot swappable, the system may be able to provide uninterrupted service in the presence of validation. Replication with validation also allows testing services to be deployed on a separate node, thereby removing any computational and storage overhead from the autonomic system.

D. Auxiliary Test Services and Test Knowledge Sources

High-level test coordination within the self-testing framework is supported by auxiliary test services (ATS) and test knowledge sources, shown at the top-right of Figure 3. The ATS component provides Orchestrating TMs with access to external or third-party automated test support tools; and implements facilities for manual test management. Orchestrating TMs interact with the ATS to configure test support tools such as code coverage profilers, performance analyzers, and automated test drivers. The ATS component supplements the services offered by these support tools and tailors them to the specific testing needs of the autonomic system.

The manual manager implementation of the autonomic system interfaces with the ATS to provide administrators with direct access to artifacts stored in test knowledge sources. The ATS will enable administrative functions such as updating validation policies, viewing test logs, managing test cases, and collating test histories through the management console of the autonomic computing system. In addition, administrators will be able to choose for certain testing tasks to require human intervention. For example, upon determining that testing is required for a

particular managed resource, the Orchestrating TM could send a message to the management console requesting the administrator to manually formulate the test plan for this resource. Other interactive administration services provided through the ATS may include defect tracking and test scenario walkthroughs.

E. Addressing Challenges

To address the many challenges of testing autonomic systems at runtime, we plan to use the proposed self-testing framework as a focal point for the investigation of various research directions that support quality assurance in these systems. One such research direction is the use of formal specifications to dynamically generate test sequences, test oracles, and test data for system components [16]. Embedding formal specifications within autonomic system components would 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 [17], which has been identified as one of the major research challenges of autonomic computing [5]. Such visualization systems could be used to interactively guide the administrator through test scenarios and during system debugging.

Regarding the problem of testing autonomic components in the presence of unforseen circumstances, mechanisms for policy-based risk analysis and trust [18] could be incorporated into the self-testing framework. Testing requirements could then be based on measurements of the estimated risk of interactions with unknown entities, or under unexpected environmental conditions. Furthermore, a risk-based strategy to regression test selection [19] could be adopted to identify test cases which would be appropriate for testing high-risk components.

IV. Test Managers: Component Interactions

Similar to the components in autonomic managers, the MAPE components in TMs implement intelligent closed loops of control. However, the intelligent control loops in TMs focus on self-testing rather than self-management. In this section, we provide descriptions of the interactions between the components of our self-testing framework in the context of these intelligent control loops. Component interactions are described at two levels of granularity. At the higher level, we discuss how the internal components of Orchestrating TMs facilitate the coordination of testing activities among multiple framework components. At the lower level, we present step-by-step details on how the internal components of Touchpoint TMs interact to accomplish low-level testing tasks on managed resources. A high-level algorithm for validating change requests is also provided in this section.

A. Interactions within Orchestrating TMs

Figure 4 illustrates how the intelligent control loops of Orchestrating TMs implement high-level test coordination

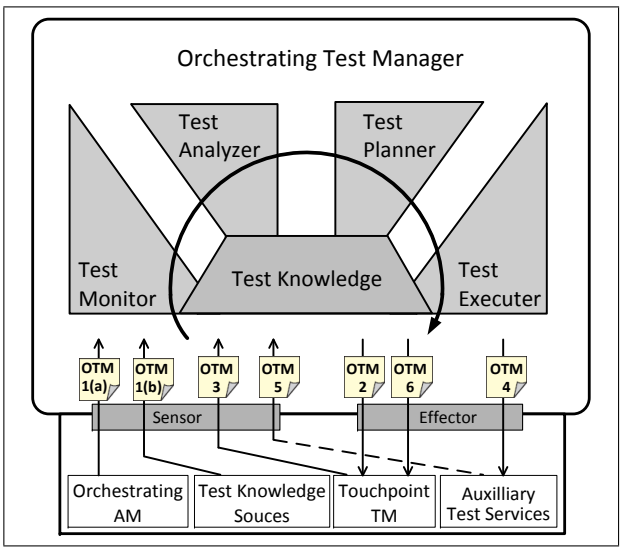

Figure 4. Intelligent closed loops of control in Orchestrating TMs.

activities. Test coordination involves complex interactions between Orchestrating TMs and multiple components of the integrated framework as shown at the bottom of Figure 4. Orchestrating TMs can invoke the sensor and effector mechanisms of Orchestrating AMs, test knowledge sources, Touchpoint TMs, and auxiliary test services. Information gathered from these framework components is then passed through the MAPE functions of the Orchestrating TM to achieve various testing objectives; as indicated by the lines labeled `OTM` that enter and exit through the sensors and effectors in Figure 4.

Orchestrating TMs will continuously poll the sensors of Orchestrating AMs (`OTM 1a`) to detect when the autonomic system requires validation services. The test knowledge sources component may also be monitored (`OTM 1b`) to detect whenever a human administrator manually updates a validation policy. Either of these events can be handled by an intelligent control loop which uploads the appropriate validation policy into Touchpoint TMs (`OTM 2`) to instantiate the new or requested testing functionality. During testing, Touchpoint TMs may be monitored to detect when they require support tools such as code coverage profilers to be configured (`OTM 3`). Configuration of the support tools could then be initiated by invoking the effector of the auxiliary test services component (`OTM 4`). Once the support tool configuration completes, this event would be detected (`OTM 5`) by the Orchestrating TM so that a notification message could be sent to the Touchpoint TM (`OTM 6`).

Other orchestrating interactions, not shown in Figure 4, include receiving notifications from Touchpoint TMs as to whether or not validation passed (`OTM 7a`) or if there was inadequate test coverage (`OTM 7b`); and responding to those notifications by either terminating the testing process (`OTM 8a`) or continuing testing (`OTM 8b`) in an attempt to improve test coverage.

B. Interactions within Touchpoint TMs

Figures 5(a) and 5(b) show two intelligent loops of control within Touchpoint TMs. The first loop traces the

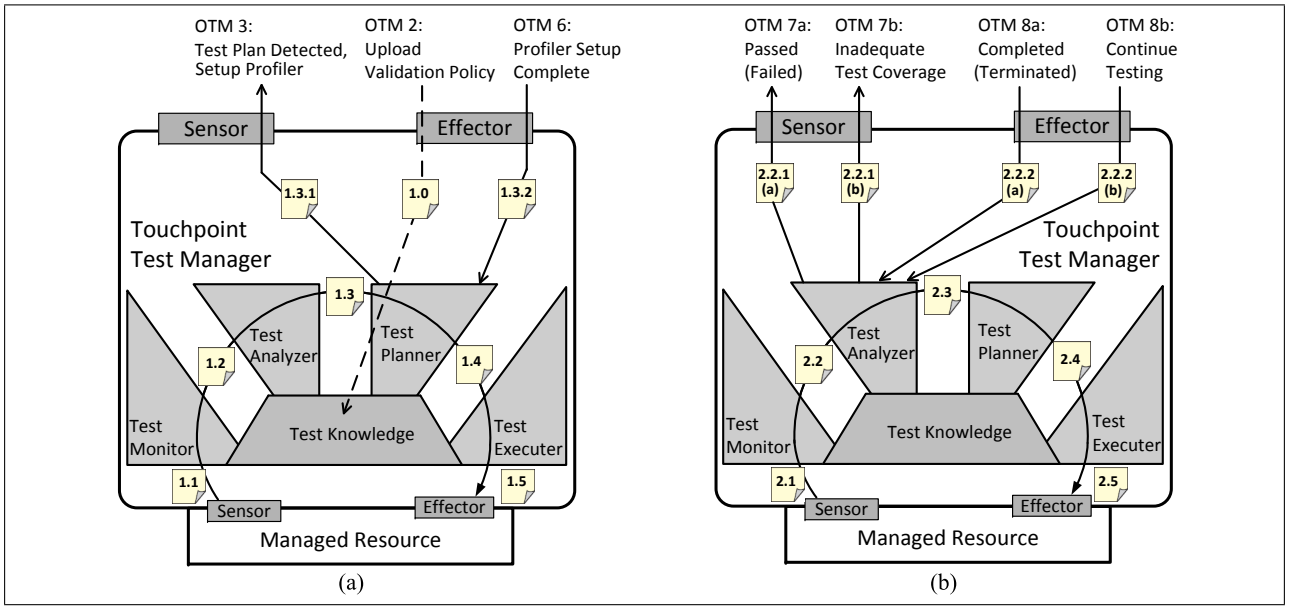

Figure 5. Component interactions through two intelligent closed control loops (a) and (b) in Touchpoint TMs.

arc labeled with artifacts `1.1` through `1.5` in Figure 5(a), and the second loop traces the arc labeled with artifacts `2.1` through `2.5` in Figure 5(b). As previously stated, the operations of Touchpoint TMs may 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. Figure 6 shows a high-level algorithm which corresponds to how the intelligent control loops of Touchpoint TMs and Orchestrating TMs work together to validate changes to managed resources. The discussion for the rest of this subsection provides details for the relationships between Figures 4, 5, and 6.

The Orchestrating TM first sends a validation policy for the managed resource to a Touchpoint TM, which loads it into the test knowledge component (`1.0`). The Touchpoint TM automatically invokes the test monitor to retrieve the new structure of the managed 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 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 request for a new test plan (`1.3.1`) has occurred and intercepts the closed loop to set up a code coverage profiling tool. Once the code profiler has been set 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 determines whether to 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, then the behavior of the second loop from `2.3` to `2.5` in Figure 5(b) is the same as the first loop from `1.3` to `1.5` in Figure 5(a).

```
Begin ValidateChangeRequest
1.0  Load Validation Policy                    (OTM 2)

1.1  Retrieve New Structure
1.2  Perform Test Suite Analysis
1.3  Request to Create Test Plan
   // External Monitored Event
     1.3.1  Test Plan Created                  (OTM 3)
   // Corresponding Actuated Event
     1.3.2  Finalize Test Plan                 (OTM 6)
1.4  Execute Test Plan
1.5  Run Test Cases

2.1  Retrieve Test Results
2.2  Perform Post Test Evaluation
   // External Monitored Events
     2.2.1a  Tests Passed or Failed            (OTM 7a)
     2.2.1b  Inadequate Coverage               (OTM 7b)
   // Corresponding Actuated Events
     2.2.2a  Stop Testing goto End             (OTM 8a)
     2.2.2b  Continue Testing goto 1.2         (OTM 8a)

End ValidateChangeRequest
```

Figure 6. Change request validation algorithm for managed resources.

V. PROTOTYPE: AN AUTONOMIC CONTAINER

Due to the lack of freely available autonomic systems, we were unable to locate an application that would allow the seamless integration of a self-testing framework. Therefore, to show the feasibility of our approach, we developed a prototype of an application with self-management capabilities. Our prototype is based on the concept of an *autonomic container* [20], which is a data structure with self-configuration and self-testing capabilities. However, we extend the work presented in [20] by implementing the container as a distributed system with the added feature of self-protection. The current version of the autonomic container therefore provides data storage services to remote users, while performing dynamic self-configuration, self-protection, and self-testing.

The prototype was developed in Java using the Eclipse SDK, along with the necessary plugins and libraries for the tools to support testing. Two tools were used to support the testing effort: JUnit, a Java unit testing tool from the xUnit family of testing frameworks [21]; and Cobertura, a Java code coverage analysis tool that calculates the percentage of source code exercised by unit tests [22]. In this section we describe the main features of the autonomic container and present its top-level design. We also provide implementation details on the self-testing subsystem, and describe our test procedures and experiment setup. We then discuss the findings of this work in the context of the uses and limitations of the prototype.

A. Main Features and Top-Level Design

The managed element of the autonomic container provides data storage services to general users through its public remote interface, while self-management features are only available to autonomic managers via its touchpoint interface. The underlying data structure for the prototype is implemented as a stack, and the following services are defined in its remote interface: `login`, `logout`, `push`, `pop`, `isFull`, and `isEmpty`.

The self-configuration and self-protection features of the prototype dynamically execute changes on the stack based on predefined symptoms. Self-configuration occurs when the stack reaches or exceeds 80% of its full capacity, and involves increasing the stack capacity to anticipate depletion of available storage space. The self-protection feature of the prototype safeguards the stack from failures that may be caused by repeated overflow or underflow conditions. Self-protection occurs when the number of stack exceptions thrown by a particular user during a session exceeds 3, and results in the associated user account being disabled.

The design of the prototype is based on the replication with validation strategy, and therefore a copy of the stack is maintained by the autonomic system exclusively for testing purposes. The top-level package for the self-testing autonomic container, labeled `edu.fiu.strg.STAC` in Figure 7, contains all the nested packages and libraries used to develop the prototype. The four main packages are as follows: (1) `mResources` – provides access to

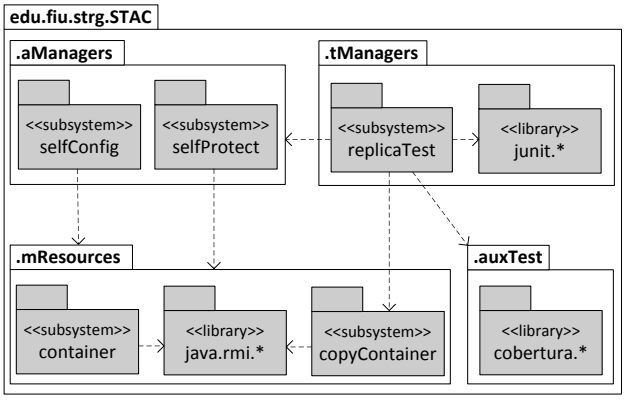

Figure 7. Top-level design of self-testing autonomic container.

the actual stack resource being managed and its copy; (2) `aManagers` – implements autonomic managers that perform dynamic self-configuration and self-protection of the stack; (3) `tManagers` – implements test managers that validate self-configuration and self-protection changes using the stack copy; and (4) `auxTest` – provides an interface to a test support tool that analyzes program code for test coverage. The packages described in (1) and (2) represent the autonomic system, while those described in (3) and (4) comprise the self-testing framework.

The aforementioned packages are composed of subsystems which provide the core functionality of the prototype. Within the `mResources` package, access to the subsystems for the remote stack `container`, and its replica `copyContainer`, is achieved by using the Java Remote Method Invocation [23] system (indicated by the dependencies on `java.rmi` in Figure 7). The package `aManagers` contains two subsystems, `selfConfig` and `selfProtect`, which implement intelligent closed loops of control for self-configuration and self-protection respectively. These two subsystems are responsible for executing change plans on the copy of the stack prior to validation, and on the actual stack if validation is successful. The package `tManagers` consists of a single subsystem `replicaTest` that coordinates the end-to-end testing activity, including the execution of parameterized test cases developed using the `junit` library. Lastly, the `auxTest` package contains classes that implement customized test services for code profiling, and automate the analysis of test coverage reports generated by the `cobertura` library.

B. Test Procedures and Setup Environment

To simulate the behavior of the architectural model of the integrated framework depicted in Figure 3, the autonomic system and self-testing framework were implemented as separate threads within a remote server application. Client programs were then developed to automatically invoke the public interface of the autonomic container, thereby emulating the actions of remote users. Both the server and client application programs were run on the Eclipse 3.2 platform using the Java Runtime Environment (JRE) 5.0 Update 12.

The client programs were designed to operate in a manner that would induce the self-management features

of the autonomic container. For example, a client was designed to login to the server and push random integers onto the stack until it exceeded 80% of its full capacity, thereby triggering dynamic self-configuration of the stack. We then observed as the self-testing framework validated the self-configuration change request, and examined the results produced during this pass of the simulation. A similar technique was used to setup a testing scenario for the self-protection feature of the prototype.

The initial test suite for the autonomic container consisted of 24 test cases written in JUnit 3.8.1. The test cases were developed using a combination of test strategies including: boundary, random, and equivalence partitioning [10]. A parameterization technique was used in test cases that addressed variable properties of the container such as stack capacity and user accounts. The validation policy for the Touchpoint TM required 100% pass rate for all test cases executed, and at least 75% branch and statement coverage. Cobertura 1.8 was used to evaluate test coverage, and generate reports in extensible markup language (XML) format.

C. Self-Test Subsystem Implementation

As previously mentioned the self-test subsystem of the prototype, `replicaTest` in Figure 7, contains managers that coordinate testing activities. These test managers, as well as the autonomic managers for self-configuration and self-protection, are based on a reusable manager design depicted by the package `edu.fiu.strg.ACSTF.manager` in Figure 8. This package is the first component developed for an *Autonomic Computing Self-Testing Framework (ACSTF)*. The design uses synchronized threads, remote method calls, generics, and reflection in Java [23] to implement the MAPE functions of the manager and the knowledge component.

The knowledge repository of the manager, represented by the class `KnowledgeData` in Figure 8, realizes the interface `KnowledgeInterface` that is used by the MAPE functions to access shared information. Queues to hold newly generated change requests and change plans to be executed have also been incorporated into the knowledge component. When a manager is instantiated, the `KnowledgeCoordinator` class loads a policy file containing the following information: (1) the fully qualified class name of the touchpoint object to be used for sensing and effecting; (2) the method name of the sensor function to be polled by the monitor; (3) a set of symptoms defined by relational mappings between the variables of the touchpoint and desired values; and (4) a set of executable change plans represented by sequences of effector methods to be executed and their actual parameters.

All high-level policies for the prototype are stored in XML 1.0 format, using the ISO-8859-1 encoding standard. Each policy contains name and version attributes which distinguish it from other policies and facilitate automation with respect to upgrades. A snippet of the

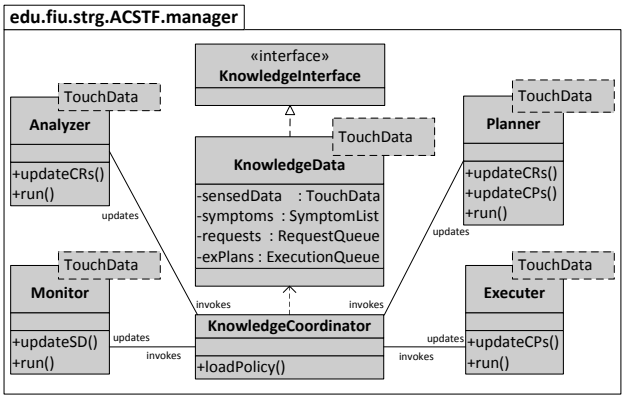

Figure 8. Generic design of autonomic and test managers.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<policy name="OTM Self-Test" version="1.0">

  <monitor touchpackage="edu.fiu.strg.STAC"
           touchclass = "OTMTouchData"
           sensormethod="getCoordState"/>

  <analyzer>
     <symptom sid="OTM_ST_001">
        <mapping var="requiresSC"
                 op ="=="
                 val="true" />
     </symptom>
  </analyzer>

  <planner>
     <plan pid="OTM_ST_001">
        <action effector="dequeueReqSC"/>
        <action effector="copyChangeSC"/>
        <action effector="setupTouchTM"/>
     </plan>
  </planner>

</policy>
```

Figure 9. Snippet of validation policy for the Orchestrating TM.

validation policy used by the Orchestrating TM of the prototype is shown in Figure 9. Policies are divided into three major sections represented by `<monitor>`, `<analyzer>`, and `<planner>` tags. Each section contains the portion of knowledge that will be primarily used by the component corresponding to its tag name. For example, combining the values of the attributes `touchpackage`, `touchclass`, and `sensormethod` of the `<monitor>` tag in Figure 9, produces the fully qualified method name of the sensor function used by the monitor of the Orchestrating TM. This method collects state information on the two autonomic managers, auxiliary test services, and Touchpoint TM to coordinate testing. Polling is achieved by first passing the fully qualified class name `edu.fiu.strg.STAC.OTMTouchData` to the monitor as the template parameter `TouchData`, shown in Figure 8. The monitor then uses reflection to instantiate the class and continuously invokes `getCoordState`.

The object returned from the invocation of the sensor method is compared with one or more predefined symptoms, which were loaded from the `<analyzer>` tag of the policy. Figure 9 shows a test symptom with its

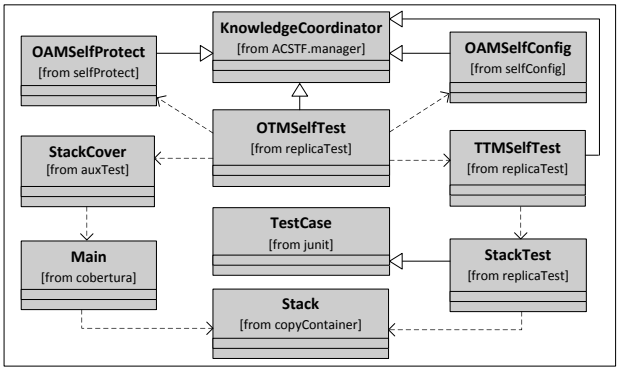

Figure 10. Class diagram of self-test and associated components.

symptom identifier (sid) attribute set to "OTM_ST_001". This symptom consists of a single mapping between the touchpoint state variable "requiresSC" and the value "true", using the relational comparison operator "==". The requiresSC variable is defined in terms of the state of the request queue of the Orchestrating AM for self-configuration, and evaluates true whenever a change request for self-configuration is pending. Therefore, if a self-configuration change request is generated, and since this is the only relational condition required by the symptom, the manager will recognize that the symptom has been satisfied and locate an appropriate test plan. Note that a similar technique is used for specifying self-management symptoms. For example, the self-configuration policy of the prototype's Touchpoint TM contains a symptom defined by the relation "usedSpacePercent >= 80". Our prototype design is highly extensible as it allows multiple variable mappings to be defined for a single symptom, and multiple symptoms to be defined for any manager.

Test symptoms are associated with test plans which provide remedies as sequences of low-level testing tasks. Test plans consist of a plan identifier (pid) attribute that is logically linked to a test symptom identifier, along with one or more actions. Figure 9 shows a test plan with pid "OTM_ST_001", which corresponds to the previously mentioned test symptom for detecting requests for self-configuration. The actions of the test plan consist of the effector methods dequeueReqSC, copyChangeSC, and setupTouchTM, which dequeue the self-configuration change request; implement the change on the stack copy; and set up the Touchpoint TM for change validation. Actions may also contain one more parameter tags (not shown in Figure 9) that provide the actual parameters to be used when executing the effector methods. In a similar fashion to the monitor, the executor component uses Java reflection to invoke the effector methods with the specified parameters.

Figure 10 shows the minimal class diagram of the self-test subsystem, along with classes from other communicating subsystems. The four manager classes OTMSelfTest, TTMSelfTest, OAMSelfConfig, and OAMSelfProtect, all extend the KnowledgeCoordinator class from the ACSTFs generic manager component. The OTMSelfTest class implements MAPE functionality of the Orchestrating TM that manages the Touchpoint TM TTMSelfTest, and monitors the two Orchestrating AMs OAMSelfConfig and OAMSelfProtect for newly generated change requests. Test cases for the stack were developed by extending the JUnit class TestCase and stored in the class StackTest. The class StackCover provides functionality for dynamically generating and executing a batch file, which sets the Main class of Cobertura to instrument the Stack class during testing.

D. Evaluation

The main objective of developing the prototype was to validate the architectural model of the integrated framework shown in Figure 3. Although the prototype only implements a minimal autonomic system, it provides evidence to support that the replication with validation strategy is feasible. A mutation testing technique was used to evaluate the prototype with respect to its ability to detect faulty change requests for managed resources. Mutation operators were applied to the stack class to create mutant stacks, which were used to assess the effectiveness of the prototype's self-testing framework.

Our evaluation involved analyzing correct and incorrect (mutant) change request scenarios for self-configuration and self-protection of the stack. The scenario for incorrect self-configuration was achieved by altering the method that resizes the stack to cause a decrease rather than an increase in the stack capacity. Similarly, incorrect self-protection of the stack involved altering the method that disables a specific user account to enable rather than disable the user. The self-testing framework was then allowed to operate on the original version of the stack, and the two mutants with the faulty methods.

In the correct change request scenarios for self-configuration and self-protection, all of the tests passed and the following code coverage measurements were recorded: self-configuration – 85% branch coverage, 100% statement coverage; and self-protection – 88% branch coverage, 100% statement coverage. The scenarios for the incorrect changes each produced two test case failures, and hence the code coverage measurements have been omitted. Our mutation analysis produced favorable results as validation passed for the correct change request scenarios, but failed for the incorrect ones. Therefore, the self-testing framework would have prevented undesirable and potentially harmful changes to the managed resource of the autonomic system.

Building the prototype provided us with insight on the scope of the self-test subsystem in terms of the operations that should be performed by the autonomic system and the self-testing framework. However, the current version of the prototype utilizes a static lookup of predefined test plans and therefore does not adequately address the need for dynamic test planning in autonomic systems. Furthermore, the prototype only implements the replication with validation strategy and is therefore limited with respect to adaptive systems.

VI. Related Work

Self-testing as an implicit feature of autonomic computing systems has received little attention in the research community. The research focus of autonomic computing has been on the major self-management properties of self-configuring, self-healing, self-optimizing and self-protecting. The pioneers of autonomic computing systems [4], [6] clearly state that one of the major challenges is validating the correctness of such systems during development and after changes have been made at runtime. It is important to note however that aspects related to the validation approaches presented in the paper have been studied in the literature. We now compare two such aspects, self-testing software/components and dynamic adaptation, along with the authors' previous work to the work presented in this paper.

Several researchers have investigated the notion of self-testing software [24]–[28]. Blum et al. [24] 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. [29] 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, which can be used to test components after an autonomic change has been implemented.

The work by Le Traon et al. [27] 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. 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.

Zhang et al. [8], [15] present an approach that formalizes the behavior of adaptive programs using state machines and Petri nets, respectively. The approach separates the adaptative behavior from the non-adaptive behavior, thereby making the models easier to specify and verify using automated techniques. The contributions of the work by Zhang and Chen [15] 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.

The work presented in this paper is an extension of the work done by King et al. [7] and Stevens et al. [20]. King et al. proposed a framework that dynamically validates change to autonomic systems. The framework includes self-testing as an implicit characteristic to support self-management in autonomic systems. Stevens et al. [20] introduced the notion of a self-testing autonomic container that uses the self-testing framework proposed by King et al. [7]. The autonomic container uses the replication with validation approach to testing the autonomic container. We extended the initial work in [7] by: (1) augmenting the self-testing framework with an auxiliary test services component and test knowledge sources to assist high-level test coordination and facilitate manual test management; (2) explicitly illustrating of how the test managers interact with these new components; and (3) stating research directions that address some of the key challenges faced by this work. The prototype presented in this paper extended the autonomic container in [7], [20] to include both dynamic self-configuration and self-protecting features. In addition, the container was implemented as a distributed system using a client-server architecture. Finally, additional details of the implementation were presented including the generic manager design, and the structure of the policies used by the managers.

VII. Conclusion

In this paper we presented an integrated self-testing framework for autonomic computing systems based on two strategies: safe adaptation with validation, and replication with validation. Our framework extends the current architecture of autonomic systems to include self-testing as an implicit characteristic. We developed a prototype of an autonomic computing system which incorporates self-testing. Clearly many challenges still remain, however, this work aims to stimulate the convergence of a set of testing and development methodologies that facilitate the construction of dependable autonomic systems.

Acknowledgment

This work was supported in part by the National Science Foundation under grants IIS-0552555 and HRD-0317692. The authors would like to thank Djuradj Babich,

Jonatan Alava, Ronald Stevens, and Mitul Patel for their contributions to this work.

REFERENCES

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

[2] Enterprise Managment Associates, "Practical autonomic computing: Roadmap to self managing technology," IBM, Boulder, CO, Tech. Rep., Jan. 2006.

[3] IBM Autonomic Computing Architecture Team, "An architectural blueprint for autonomic computing," IBM, Hawthorne, NY, Tech. Rep., June 2006.

[4] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–52, January 2003.

[5] J. O. Kephart, "Research challenges of autonomic computing," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 15–22.

[6] H. A. Muller, L. O'Brien, M. Klein, and B. Wood, "Autonomic computing," Carnegie Mellon Univeristy and Software Engineering Institute, Tech. Rep., April 2006.

[7] T. M. King, D. Babich, J. Alava, R. Stevens, and P. J. Clarke, "Towards self-testing in autonomic computing systems," in *ISADS '07: Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 51–58.

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

[9] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[10] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit testing coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, December 1997.

[11] D. Mosley and B. Posey, *Just Enough Software Test Automation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[12] Mosaic, Inc., "Staffing your test automation team," Mosaic, Inc., Chicago, IL, Tech. Rep., April 2002.

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

[14] 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.*, vol. 10, no. 2, pp. 184–208, 2001.

[15] 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*. New York, NY, USA: ACM Press, 2006, pp. 371–380.

[16] E. Bernard, B. Legeard, X. Luck, and F. Peureux, "Generation of test sequences from formal specifications: Gsm 11-11 standard case study," *Softw. Pract. Exper.*, vol. 34, no. 10, pp. 915–948, 2004.

[17] N. Dulac, T. Viguier, N. G. Leveson, and M.-A. D. Storey, "On the use of visualization in formal requirements specification," in *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 71–80.

[18] K. Feeney, K. Quinn, D. Lewis, D. O'Sullivan, and V. Wade, "Relationship-driven policy engineering for autonomic organisations," in *POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 89–98.

[19] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-based regression test selection with risk analysis," in *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2002, p. 1.

[20] R. Stevens, B. Parsons, and T. M. King, "A self-testing autonomic container," in *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*. New York, NY, USA: ACM Press, 2007, pp. 1–6.

[21] E. Gamma and K. Beck, "JUnit 4.3.1 - Testing Resources for Extreme Programming," 2005, http://www.junit.org/index.htm (June 2007).

[22] M. Doliner, G. Lukasik, and J. Thomerson, "Cobertura," 2002, http://cobertura.sourceforge.net/ (June 2007).

[23] Sun Microsystems, Inc., "Core Java J2SE 5.0," February 2005, http://java.sun.com/j2se/1.5.0/index.jsp (Aug. 2006).

[24] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *Journal of Computer and System Sciences*, vol. 45, no. 6, p. 549595, 1993.

[25] G. Denaro, L. Mariani, and M. Pezzè, "Self-test components for highly reconfigurable systems," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, 2003.

[26] 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*. London, UK: Springer-Verlag, 1995, pp. 248–262.

[27] 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)*. IEEE Computer Society, 1999, pp. 96–107.

[28] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *J. ACM*, vol. 44, no. 6, pp. 826–849, 1997.

[29] 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)*, 2005, pp. 553–557.

Tariq M. King is currently a Ph.D. candidate at Florida International University (FIU). He received his MS and BS degrees in computer science from FIU and the Florida Institute of Technology in 2007 and 2003, respectively. His research interests include autonomic computing, model-based testing, and model checking. Tariq is a member of the ACM, IEEE Computer Society, and Software Testing Research Group (STRG) at FIU.

Alain E. Ramirez is an undergraduate student in Computer Science at FIU who is currently participating in the Summer 2007 Research Experience for Undergraduates (REU) program, which is sponsored by the National Science Foundation. He is a member of the ACM, IEEE Computer Society, and STRG.

Rodolfo Cruz is a graduate student at FIU currently pursuing an MS degree in Computer Science. He received his BS degree in computer science from FIU in 1997. Rodolfo has over 10 years experience in the software development industry, and is currently a project manager for Oceanwide USA, Inc. He is a member of the ACM, IEEE Computer Society, and STRG.

Peter J. Clarke received his Ph.D. in Computer Science from Clemson University in 2003, and his M.S. degree from Binghamton University in 1996. His research interests include software testing, software metrics, software maintenance, and model-driven development. He is currently an Assistant Professor in the School of Computing and Information Sciences at FIU. He is a member of the ACM, and IEEE Computer Society.