Safe Runtime Validation of Behavioral Adaptations in Autonomic Software

Tariq M. King[1], Andrew A. Allen[2], Rodolfo Cruz[2], and Peter J. Clarke[2]

[1] Department of Computer Science
North Dakota State University, Fargo, ND 58108, USA
tariq.king@ndsu.edu

[2] School of Computing and Information Sciences
Florida International University, Miami FL 33199, USA
{aalle004, rcruz002, clarkep}@cis.fiu.edu

Abstract. Although runtime validation and verification are critical for ensuring reliability in autonomic software, research in these areas continues to lag behind other aspects of system development. Few researchers have tackled the problem of testing autonomic software at runtime, and the current state-of-the-art only addresses localized validation of self-adaptive changes. Such approaches fall short because they cannot reveal faults which may occur at different levels of the system. In this paper, we describe an approach that enables system-wide runtime testing of behavioral adaptations in autonomic software. Our approach applies a dependency-based test order strategy at runtime to facilitate integration and system-level regression testing in autonomic software. Since validation occurs on-line during system operations, we perform testing as part of a safe approach to adaptation. To investigate the feasibility of our approach, we apply it to an autonomic communication virtual machine.

Keywords: Validation, Self-Testing, Autonomic Software, Adaptation

1 Introduction

Autonomic computing (AC) describes systems that manage themselves in response to changing environmental conditions [9]. The popularity of the AC paradigm has led to an increase in the development of systems that can self-configure, self-optimize, self-protect, and self-heal [7, 9]. These self-* features are typically implemented as *Monitor-Analyze-Plan-Execute* (MAPE) loops in autonomic managers (AMs). AMs monitor the state of managed computing resources, and analyze the observed state information to determine if corrective action is required. When undesirable conditions are observed, the AM formulates and executes a plan to remedy the situation. There are two levels of AMs: *Touchpoint* – directly manage computing resources through sensor and effector interfaces, and *Orchestrating* – coordinate the behavior of multiple AMs [7].

Some self-* changes may involve adapting or updating system components at runtime, a process referred to as dynamic software adaptation [24]. Dynamic

adaptation in autonomic software raises concerns regarding reliability, since new faults may be introduced into the system by runtime changes. Runtime validation and verification are therefore expected to play a key role in AC systems. However, a 2009 survey on the research landscape of self-adaptive software stated that "testing and assurance are probably the least focused phases in engineering self-adaptive software" [18]. Although this has improved slightly within the last two years, it is evident that research on testing autonomic software continues to lag behind other areas.

Few researchers have tackled the problem of runtime testing in autonomic software [3, 10]. King et al. [10] introduce an implicit self-test characteristic into autonomic software, referred to as *Autonomic Self-Testing* (AST). Under AST, the existing MAPE implementation is used to define test managers (TMs), which monitor, intercept, and validate the change requests of autonomic managers. TMs can operate according to two strategies: *Replication with Validation* (RV) – tests autonomic changes using copies of managed resources; and *Safe Adaptation with Validation* (SAV) – tests autonomic changes in-place, directly on managed resources. Until now, investigation into AST has concentrated on developing and evaluating prototypes that implement localized runtime testing, according to the RV strategy [11, 15, 20]. Developing these prototypes revealed that RV is a viable AC runtime testing technique, which can be employed when copies of managed resources are easily obtainable for testing purposes [15, 20]. Details on the approach by Da Costa et al. [3] are provided in the related work section.

In this paper, we extend previous work on AST by overcoming two of its current limitations. Firstly, we address the need for system-wide validation in autonomic software through the description of a runtime integration testing approach. Our approach views the autonomic system as a set of interconnected, *Self-Testable Autonomic Components* (STACs), and emphasizes operational integrity during the runtime testing process. Secondly, we investigate AST of a real-world autonomic application for which it is expensive to maintain test copies of managed resources. Our application motivates the need for SAV and system-wide AST, and is used as a platform for investigating their feasibility. The rest of this paper is organized as follows: Section 2 provides related work. Section 3 describes our testing approach. Section 4 presents a detailed design for STACs. Section 5 discusses the prototype, and in Section 6 we conclude the paper.

2 Related Work

Runtime validation and verification as an integral feature of autonomic software has received little attention in the research literature [3, 10, 25, 26]. The approach described by Da Costa et al. [3] is most closely related to our work. It presents the Java Self-Adaptive Agent Framework + Test (JAAF+T). JAAF+T incorporates runtime validation into adaptation agents by directly modifying their MAPE control loops to include a test activity [3]. Self-testing under JAAF+T is localized within agents, and as described is not feasible at the integration and system

levels. Furthermore, the authors do not address how the integrity of system operations is enforced during the self-test activity.

Stevens et al. [20] developed a prototype of an autonomic container to demonstrate the idea of runtime testing in autonomic software. An autonomic container is a data structure with self-configuring and self-testing capabilities. The purpose of developing the application was to provide a simplistic, lightweight model of how autonomic software operates, and use it to investigate the Replication with Validation (RV) strategy [10]. Ramirez et al. [15] later extended the autonomic container, and applied it to the problem of short-term job scheduling for an operating system. Like its predecessor, the self-test design of the autonomic job scheduler was based on the RV strategy. To the best of our knowledge, the autonomic communication virtual machine presented in this paper is the first prototype to implement Safe Adaptation with Validation (SAV) [10].

Zhang et al. [25] proposed modular model checking for runtime verification of adaptive systems. They harness a finite state machine to check transitions between variations points in the software. To address the problem of state space explosion, they confine model checking to those aspects of the system affected by the change. In a similar fashion, we use dependency analysis to reduce the number of test cases that need to be re-run after an adaptive change. The work by Zhao et al. [26] presents a model-based runtime verification technique that can be applied to autonomic systems. More specifically, their approach targets component-based, self-optimizing systems that dynamically exchange components. The on-line model checker by Zhao et al. [26] is interleaved with the execution of the autonomic system, and shares technical challenges with our work. In general, model checking may offer a viable alternative to the research problem in cases where runtime verification is preferred over testing.

3 Testing Approach

Autonomic software may be viewed as a set of composable, interacting, autonomic components that provide localized self-management through sensors and effectors, while being environmentally aware and connected via a system-wide control loop [16]. The component-based perspective of autonomic software is widely accepted and cited in the literature [12, 14]. Furthermore, the increasing trend towards Service-Oriented Architectures, Web and Grid Services, and Cloud Computing suggests that the component-based paradigm represents a pragmatic approach for building these next-generation software systems.

In this section, we describe an approach that facilitates system-wide validation of behavioral adaptations in component-based autonomic software. The boundaries of self-testing are delineated by analyzing dependency relationships between the adaptable components and other components in the system. Any component that invokes an adaptable component is considered to be within the firewall of change, and is therefore made self-testable. We now provide an overview of our approach, and describe its major steps through a workflow, algorithm, and state-based model.

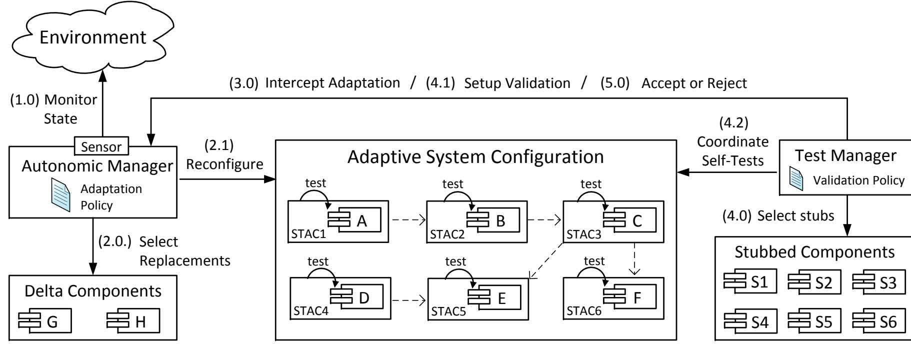

Fig. 1. System-Wide Validation of Behavioral Adaptations using STACs

3.1 Overview

Our approach views the adaptive portion of the autonomic software as a set of self-testable autonomic components (STACs). STACs encapsulate the dynamically replaceable baseline components of the system, which represent the software services being managed (e.g., a local service, web service, or off-the-shelf component). If an autonomic change replaces a baseline component with a delta component that has never been used within the context of the application, self-testing is performed as part of its integration into the system.

Figure 1 presents our system-wide architectural perspective for autonomic software using STACs. Our architecture focuses on validation scenarios in which a baseline component is replaced by some delta component that conforms to the same service interface. Such behavioral adaptations are frequently realized in the implementations of adaptive software, and are directly supported by many component-based software development frameworks [22]. Boxes `A` through `F` in Figure 1 represent six baseline components of the autonomic software, which have been made self-testable by enclosing them in `STAC1` through `STAC6`. The arcs labeled `test` denote the ability of each STAC to perform runtime testing on the application services of its baseline component. Predefined test cases for the application services are be stored within the STAC. Delta components (bottom-left of Figure 1) are represented by a set of components that are dynamically discoverable via a service registry or knowledge repository.

The workflow of our testing approach, as relates to Figure 1, is described as follows (starting from the top-left): An Orchestrating Autonomic Manager (AM) continuously monitors the system environment `(1.0)` and analyzes its state according to a predefined adaptation policy. In response to environmental changes, the AM selects the replacement components `(2.0)`, and dynamically reconfigures the system `(2.1)`. If runtime validation is required, an Orchestrating Test Manager (TM) intercepts adaptation and generates a test plan `(3.0)`. The TM then selects any stubs `(4.0)` that may be required for testing, and passes them to the AM in a request to set up the validation process `(4.1)`. Lastly, the

TM coordinates a series of self-tests (`4.2`) using the STACs, and evaluates the results to determine if the adaptive change should be accepted or rejected (`5.0`).

We have refined the self-test coordination activity, identified in workflow step `4.2`, into Algorithm A.1. Our algorithm incorporates a graph-based integration test order strategy [2].

Algorithm A.1. Self-Test Coordination using STACs

For each delta component D_i being integrated:

1. Replace the baseline component targeted in the adaptation with D_i.
2. If D_i calls other components, replace the callees with stubbed components; and invoke a self-test on D_i.
3. Generate an integration test order (ITO) for D_i to minimize the number of stub re-configurations required during D_i's integration.
4. Use the ITO from Step 3 to invoke a series of adaptations and self-tests until all of the previously replaced callee components have been integrated.
5. Perform dependency analysis to identify the components that call D_i; and invoke self-tests on them in reverse topological order.

3.2 Illustrative Example

To illustrate the behavior of the algorithm, we now apply it to an adaptation scenario. Our scenario involves the baseline component `C` from `STAC3` in Figure 1 being replaced by the delta component `H`. In this case the algorithm produces the following list of actions:

Action 0. Replace `C` in `STAC3` with `H`
Action 1. Replace `E` in `STAC5` with `S5`
Action 2. Replace `F` in `STAC6` with `S6`
Action 3. Execute `STAC3.selftest`
Action 4. Replace `S5` in `STAC5` with `E`
Action 5. Execute `STAC3.selftest`
Action 6. Replace `S6` in `STAC6` with `F`
Action 7. Execute `STAC3.selftest`
Action 8. Execute `STAC2.selftest`
Action 9. Execute `STAC1.selftest`

During Action 0 the system replaces the baseline component `C` with the delta component `H`, and the self-testing process is initiated. Actions 1–3 set up and execute self-tests on `H` using stubs `S5` and `S6`. The stubbed configuration allows the behavior of `H` to be validated in isolation, as depicted by the arc labeled `Unit` and the stereotypes labeled <<`stubbed`>> in Figure 2a. Actions 4 and 5 remove the `S5` stub and validate the interactions between `H` and one of its actual dependents `E` (Figure 2b). Similarly, Actions 6 and 7 realize an integration test between `H` and its dependent `F` through replacement of the stub `S6` (Figure 2c). At this point the system has reached its target configuration. Actions 8 and 9

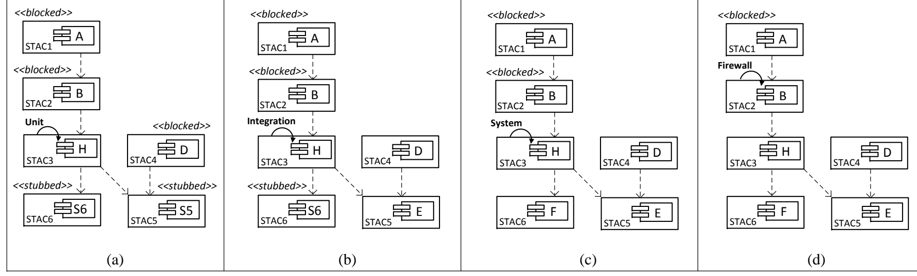

Fig. 2. Illustrative Example of Unit, Integration, System, and Firewall Self-Tests.

conclude the self-testing process with the firewall regression test phase which validates each caller component affected by H's integration. This involves first performing a self-test on component B (Figure 2d), and then component A. The stereotype <<blocked>> in Figure 2 indicates the partial disablement of caller components for the purpose of runtime safety.

3.3 State-Based Model for SAV

Recall that Safe Adaptation with Validation (SAV) tests adaptive changes to autonomic software in-place, during the adaptation process [10]. The strategy is based on the idea of *safe adaptation* [24], which ensures that the adaptation process does not violate dependencies between components, nor interrupt critical communication channels. Figure 3 presents a state-based model for our system-wide testing approach according to SAV. It extends the model by Zhang et al. [24] with: (1) test setup activities via a refinement of the `resetting`, `safe`, and `adapted` states; and (2) test invocation, execution, and evaluation activities by adding a new `validating` state.

Before SAV begins, the system is in the `running` state where all components are fully operational. An Orchestrating TM then sends a reset command to an Orchestrating AM, which moves the system into the `resetting` state. During the reset, the system is in partial operation as the AM disables functions associated with the adaptation target (AT), and stubbed dependencies (SD). Upon completion, the system is in a `safe` state where the adaptation target is replaced by the delta component, and the system begins `adapting for unit tests`.

Once all callees of the delta component have been stubbed, the system transitions to the `ready for unit tests` state. The TM then sends a test command and the system enters the `validating` state, where test cases are executed. If unit testing is successful, the system moves to the `adapting for integration tests` state. Stubbed callees are replaced by actual callees one at a time, with a test command issued after each replacement. In other words, the system alternates between `adapting for integration tests` and `validating`, until the interactions of callees with the delta component have been tested.

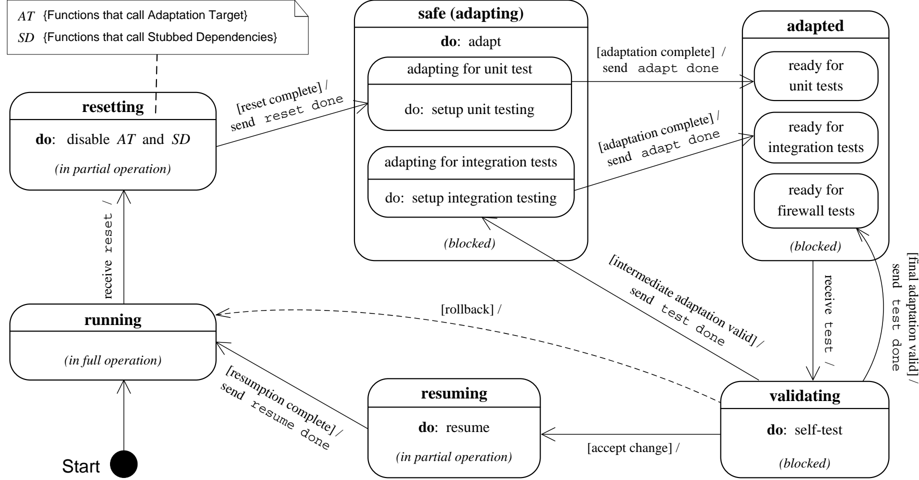

Fig. 3. State-Based Model for Safe Adaptation with Validation

After reaching the target configuration, the system moves from validating to the `ready for firewall tests` state. The TM then invokes the final set of test commands. If the firewall tests pass, the delta component is accepted and the system enters a `resuming` state. During resumption, the AT and SD functions are unblocked and then the system returns to a fully operational `running` state. At any point during unit, integration, system, or firewall regression testing, the TM can reject the adaptive change and rollback to the previous configuration (dotted transition in Figure 3).

4 Self-Testable Autonomic Components

In this section we formalize the notion of a self-testable autonomic component (STAC), and provide a detailed design to support STAC development. As depicted in Figure 4, a STAC is defined by a 5-tuple (T, A, R, I, K) where:

T is a finite set of test managers which are responsible for validating self-management changes to resource R

A is a finite set of autonomic managers, disjoint from T, which perform self-management of resource R

R is a computational or informational resource which provides application-specific services to its clients

I is a finite set of interfaces for client services (I_S), self-management (I_A), runtime testing (I_T), and maintenance (I_M)

K is a knowledge repository containing artifacts such as policies (K_P), test cases (K_T), and test logs (K_L)

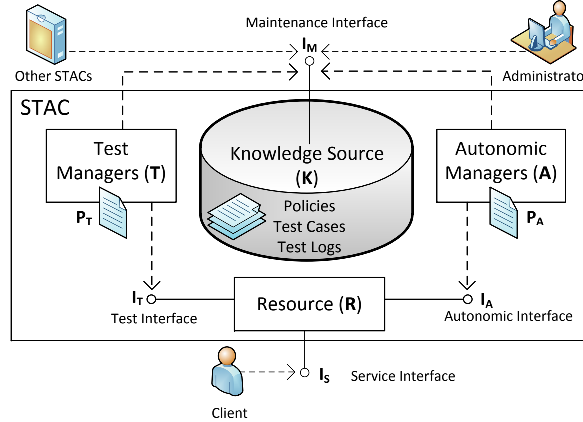

Fig. 4. Elements of a Self-Testable Autonomic Component.

4.1 Managers

A reusable design of the test managers (T) and autonomic managers (A) is shown in Figure 5. Both types of managers use a generic design that extends the work in King et al. [11] by: (1) incorporating safety mechanisms for suspending and resuming managers; (2) adding support for updating the internal knowledge via an external knowledge source, and (3) abstracting common logic out of the monitor, analyze, plan, execute (MAPE) functions.

The `GenericManager` class, shown at the top-right of Figure 5, is the main controller class that coordinates the manager's activities. The generic manager may invoke an internal knowledge component or an external knowledge source, as indicated by the interfaces `InternalKnowledge` and `KnowledgeSource` respectively. The key operations provided by the generic manager include: `activate` – sets a specified behavioral policy `p` as being active in the internal knowledge; `manage` – starts or resumes the MAPE functions; `suspend` – temporarily pauses the MAPE functions for safety or synchronization purposes; and `update` – retrieves new or updated behavioral policies from an external knowledge source.

The template parameter `Touchpoint`, represented by the dotted boxes in Figure 5, is a place-holder for the class that implements the self-management or self-test interface used by the manager. Instantiation of the `GenericManager` requires the fully qualified class name of this `Touchpoint` class, and the name of the sensor method that will be used to poll the managed resource. The package `edu.fiu.strg.STAC.manager.mape` in Figure 5 shows the detailed class design of the MAPE functions of the generic manager. Each function derives from the abstract class `AbstractFunction`, which implements common behaviors such as: (1) initialization, suspension and resumption of the function; and (2) access to data shared among the functions.

Function independence is achieved through programming language support for multi-threading, as indicated by specialization of the `Thread` library (top-left of Figure 5). Once a MAPE object is initialized, its control thread con-

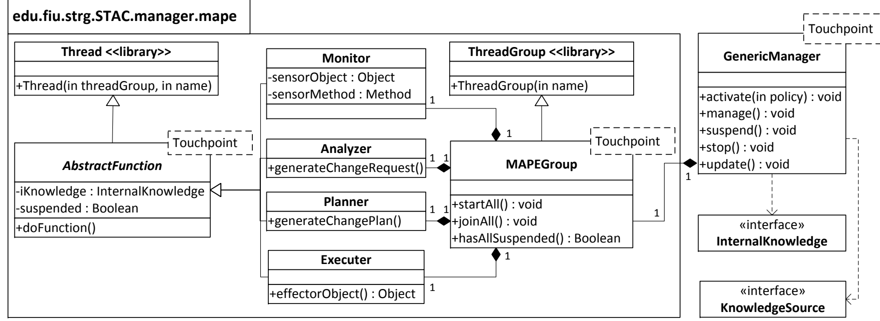

Fig. 5. Extended Design of Generic Manager with Runtime Safety Mechanisms

tinuously invokes a function-specific implementation of the abstract method `doFunction()`. For example, a monitor's doFunction implements state polling of managed resources, while an analyze doFunction compares that state against symptom information. The boolean variable `suspended` in `AbstractFunction` denotes whether or not the function has been temporarily paused. Access to data shared by the functions is through the variable `iKnowledge`, which implements the `InternalKnowledge` interface.

The concrete MAPE implementation is represented by `Monitor`, `Analyzer`, `Planner`, and `Executer` classes (center of Figure 5). Both the `Monitor` and `Executer` incorporate reflection-oriented programming techniques to avoid hard-coding the qualified method names of sensors and effectors. Sensors and effectors can therefore be updated at runtime, which allows them to be made consistent with the resource even after structural adaptations. All MAPE threads within the manager can be manipulated as a single unit. The class labeled `MAPEGroup` in Figure 5 contains the synchronization logic for initializing, suspending, and resuming a collection of MAPE threads. Some programming languages provide built-in support for thread grouping and synchronization, as indicated by the `ThreadGroup` library.

4.2 Internal Knowledge

Figure 6a shows the design of the internal knowledge of the generic manager. The `KnowledgeData` class (top-center) realizes the interface `InternalKnowledge`, which is used by the MAPE functions to access shared data. The key attributes of the `KnowledgeData` class are: `touchData` – holds the current state information of the resource R captured by the monitor function; `symptoms` – represents a set of conditional relations used by the analyze function to determine if R is in an undesirable state; and `changePlans` – contains an action, or sequence of actions, generated by the plan function in order to transition R back to a desired state.

Individual symptoms are defined by the conjunction of relations between different state variables and corresponding values. As shown at the left of Fig-

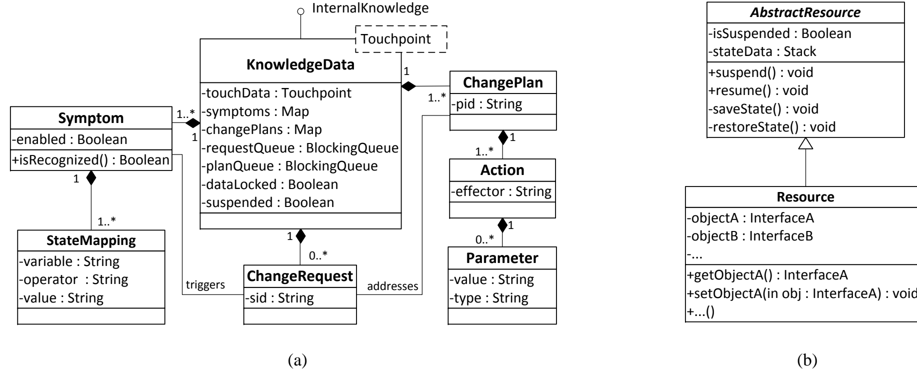

Fig. 6. (a) Design of Internal Knowledge, and (b) Managed Resources.

ure 6a, a `Symptom` is composed of one or more `StateMapping` objects. Each `StateMapping` consists of: (1) attributes that hold information on a single state variable and an associated value, and (2) a relational operator. The method `isRecognized()` of the `Symptom` class computes the truth value of each `State-Mapping`, and returns the conjunction of the results to indicate whether or not the managed resource is in an undesired state.

Recognition of a symptom triggers the generation of a `ChangeRequest` object. Change requests contain a symptom identifier `sid` that is used to lookup a `ChangePlan` to address the problem. Each `ChangePlan` is composed of one or more `Action` objects, which provides the name of the `effector` method and an optional `Parameter` list. Two queues, `requestQueue` and `planQueue`, have been incorporated into the `KnowledgeData` class to hold change requests and change plans respectively. These blocking queues use producer-consumer relationships to facilitate safe communication among the MAPE functions.

4.3 Resource

Our detailed design of the resource R considers two scenarios: (1) developers are required to build R from scratch and/or have full access to modify its source code; and (2) a commercial off-the-shelf (COTS) component, whose source code is unavailable, is to be used as the primary basis for developing R.

Developing *R* with Code Access Figure 6b provides a detailed class design for R when there are no limitations with respect to source code accessibility. Each managed resource derives from the class labeled `AbstractResource` in Figure 6b. This abstract class provides access control mechanisms at the resource-level to address our safety goal. These include the operations: `suspend`, `resume`, `saveState`, and `restoreState`.

State visibility and controllability are addressed by requiring that *getters* and *setters* be provided for all of R's private and protected variables. These include variables of both primitive and non-primitive data types. Although intrusive, this requirement ensures that test case pre- and post- conditions associated with object state can be setup and verified. To further improve the testability of R, our design emphasizes the use of dependency injection [22]. This mandates that all non-primitive data within R be declared as references to interfaces, rather than implementation classes (e.g., objects `A` and `B` in Figure 6b). Following such a design heuristic decouples the interface of each object in R from its source code, thereby allowing mock object implementations (stubs) to be easily swapped in and out of R during runtime testing.

Developing *R* without Code Access In the scenario where R is a COTS component, the adapter design pattern can be used to facilitate the implementation of the safety mechanisms specified in Figure 6b. This would involve building a wrapper class that adds suspend and resume operations to the component's existing functionality. Care should be taken when developing the wrapper so that new errors are not introduced into R during this activity.

Using COTS can impede both the manageability and testability of R since there is usually no way to break encapsulation, and gain access to the component's private or protected members. However, the complementary strategy presented by Rocha and Martins [17] can be used to improve testability of components when source code is not available. Their approach injects built-in test mechanisms directly into intermediate code of the COT. Other approaches for improving the runtime testability of COTS exist in the literature, and may be applicable to R in this scenario.

4.4 Test Interface

A class diagram of the self-test support design is shown in Figure 7. Three categories of automated testing tools have been modeled for use in STACs, represented by the interfaces: `ExecutionCollector` – applies test cases to the resource to produce item pass/fail results; (2) `CoverageCollector` – instruments the source code and calculates line and branch coverage during test execution; and (3) `PerformanceCollector` – computes the total elapsed time taken to perform test runs.

The class `SelfTestSupport` realizes the collector interfaces, and is used to store the results of testing via the attribute `testResults`. After testing completes, this data structure can be queried by TMs to gather information such as the number of test failures; total elapsed time for testing; and the percentage of line and branch coverage. TMs can then store this information in their internal knowledge as test logs, and evaluate them against the predefined test policies. The `SourceMap` class is used to configure the self-test implementation by providing information on managed resources, including the location of related source code modules and automated test scripts.

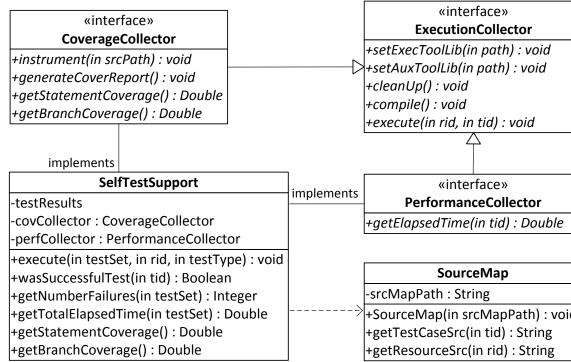

Fig. 7. Test Interface and Implementation.

5 Prototype

To investigate the feasibility of the ideas presented in this paper, we have applied system-wide AST to a self-configuring and self-healing *Communication Virtual Machine* (CVM) [4]. CVM is a model-driven platform for realizing user-centric communication services, and is the result of a collaboration between FIU SCIS and Miami Children's Hospital in the domain of healthcare.

5.1 Application Description

Allen et al. [1] leverage autonomic computing, and open-platform communication APIs, in the provision of a comprehensive set of communication services for CVM. Figure 8 shows the *Network Communication Broker* (NCB) layer of CVM. Through a series of dynamic adaptations, NCB self-configures to use multiple communication frameworks such as Skype, Smack, and Native NCB [8, 19, 23] (bottom-right). An NCB management layer exposes the communication services to its clients via a network independent API (top of Figure 8).

A layer of Touchpoint AMs directly manages these communication services and is responsible for dynamically adapting these components in the network configuration. An Orchestrating AM, labeled `OAMCommunication`, coordinates the Touchpoint AMs, and analyzes user requests stored in a call queue to determine if self-management is required. Requests for self-management can have two general forms: (1) a user's explicit request for communication that is not supported by the current configuration, thereby requiring self-configuration; and (2) the occurrence of other erroneous events such as failure of a communication service, thereby requiring self-healing.

5.2 Setup and Experimentation

Using the approach and supporting designs described in Sections 3 and 4, we incorporated self-testing into the NCB layer of CVM. A test manager (TM)

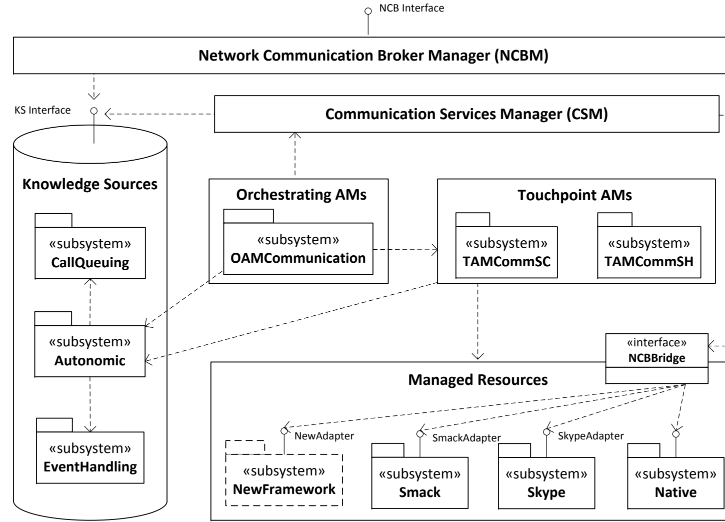

Fig. 8. Network Communication Broker (NCB) Layer of CVM

was implemented at the orchestrating-level to monitor change requests generated by `OAMCommunication`. Change requests requiring dynamic adaptation were validated in-place by a Touchpoint TM, after disabling caller functions in the Communication Services Manager and Touchpoint AMs to ensure safety. The automated tools JUnit and Cobertura were used to support the testing process [5, 6]. A total of 41 JUnit tests were developed to validate self-adaptations in the NCB, along with the necessary test drivers and stubs. Cobertura was set up to instrument the NCB implementation for statement and branch coverage.

Experiments were conducted using the CVM platform in order to gain insights into the benefits and challenges associated with the proposed approach. The experiments were designed to facilitate the observation and measurement of the: (1) effectiveness of self-tests in detecting faults and exercising the CVM implementation, and (2) impact of runtime testing on the processing and timing characteristics of the application. The Eclipse Test and Performance Tools Platform (TPTP) was used for timing test executions and measuring thread utilization during self-testing [21].

A mutation analysis technique was incorporated into the experimental design. This involved generating 39 faulty delta components by planting artificial defects into the interface and/or implementation of: (1) Skype, representing a proprietary closed-source component, and (2) Smack, representing an open-source component. 15 of the mutants were created manually, while the remaining 24 were generated automatically using MuJava [13]. A driver was written to simulate the environmental conditions necessary to induce self-configuration and self-healing of CVM using the mutant components, at which point self-testing would occur to produce the results.

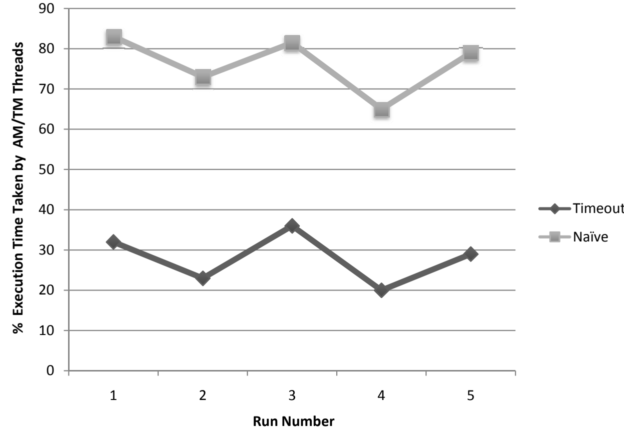

Fig. 9. Thread Utilization of Test Managers in CVM

5.3 Results

For the fault detection experiments 35 out of the 39 mutants were detected, producing a mutation score of 89.7%. Self-testing achieved 63% statement coverage and 57% branch coverage of the NCB's implementation. In addition, there was 100% method coverage of the communication service interfaces. Figure 9 provides the manager thread performance results of five experimental runs using two variants of the NCB. One variant uses a *naive* monitoring scheme that polls resources continuously, and the other applies a *timeout* based monitoring scheme that polls resource intermittently every 250ms. The data for the experimental runs were collected during a 2-way Skype AV to a 3-way Smack AV self-configuration.

5.4 Discussion

The findings of our prototype experimentation suggest that both system-wide AST and SAV are feasible. In particular, mutation analysis revealed highly favorable scores, with multiple test failures being produced at the integration and system-levels. Self-testing in the NCB successfully detected a cross section of bad mutants, and exercised a significant portion of the program structure.

As indicated by Figure 9, the initial runtime interleaving of the CVM and its self-testing process was highly biased in favor of testing. This led to observable degradation of the core CVM services, especially during the establishment of communication connections. However, introducing a small timeout into the TM threads of the NCB significantly improved the performance of the core CVM services. The timeout reduced the biased interleaving from a range of 85-65% to 35-20%, which was acceptable to CVM users.

Using safety mechanisms to debug the self-testing CVM was instrumental to the success of the project. Debugging and off-line testing of the NCB layer relied heavily on the use of suspend and resume operations within AMs and

TMs. Developing self-tests for the NCB was a significant challenge due to its large size and complexity. Some tests required making asynchronous calls to the underlying communication frameworks. However, most open-source testing tools do not support asynchronous testing, which meant that we had to implement additional programs for this purpose.

Selecting and tailoring open-source testing tools for use in autonomic software presented additional difficulties, including: (1) locating trustworthy information on tool features and configuration procedures; (2) filtering tools that would not be suitable for the problem being tackled due to their reliance on external dependencies, e.g., Apache ANT, and (3) determining which report formats would be most appropriate for automatically extracting results in a uniform manner, e.g., CSV, XML, HTML. Threats to the validity of the investigation include performance error due to the instrumentation overhead, and the lack of similar studies for verifying experimental results.

6 Conclusion

Dynamically adaptive behavior in autonomic software requires rigorous offline and on-line testing. To narrow the gap between on-line testing and other advances in autonomic computing, we presented a system-wide approach for validating behavioral adaptations in autonomic software at runtime. Emphasis was placed on enabling on-line testing at different levels of the autonomic system, as well as maintaining operational integrity at runtime via built-in safety mechanisms. A prototype of a self-testing autonomic communication virtual machine was presented, and used to discuss the feasibility, challenges and benefits of the proposed approach. Future work calls for controlled experimentation of the self-testing CVM, and investigating runtime testing of structural adaptations.

Acknowledgments

The authors would like to thank Alain E. Ramirez for his contribution to this research. This work was supported in part by the NSF under grant IIS-0552555.

References

1. Allen, A.A., Wu, Y., Clarke, P.J., King, T.M., Deng, Y.: An autonomic framework for user-centric communication services. In: CASCON '09. pp. 203–215. ACM, New York, NY, USA (2009)
2. Briand, L.C., Labiche, Y., Wang, Y.: An investigation of graph-based class integration test order strategies. IEEE Trans. Software Eng. 29(7), 594–607 (2003)
3. Da Costa, A.D., Nunes, C., Da Silva, V.T., Fonseca, B., De Lucena, C.J.P.: JAAF+T: a framework to implement self-adaptive agents that apply self-test. In: SAC '10. pp. 928–935. ACM, New York, NY, USA (2010)
4. Deng, Y., Sadjadi, S.M., Clarke, P.J., Hristidis, V., Rangaswami, R., Wang, Y.: CVM-a communication virtual machine. J. Syst. Softw. 81(10), 1640–1662 (2008)

5. Doliner, M., Lukasik, G., Thomerson, J.: Cobertura 1.9 (2002), http://cobertura.sourceforge.net/ (June 2011)
6. Gamma, E., Beck, K.: JUnit 3.8.1 (2005), http://www.junit.org/ (June 2011)
7. IBM Autonomic Computing Architecture Team: An architectural blueprint for autonomic computing. Tech. rep., IBM, Hawthorne, NY (June 2006)
8. Jive Software: Smack API (November 2008), http://www.igniterealtime.org/projects/smack/ (June 2011)
9. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–52 (January 2003)
10. King, T.M., Babich, D., Alava, J., Stevens, R., Clarke, P.J.: Towards self-testing in autonomic computing systems. In: ISADS '07. pp. 51–58. IEEE Computer Society, Washington, DC, USA (2007)
11. King, T.M., Ramirez, A., Clarke, P.J., Quinones-Morales, B.: A reusable object-oriented design to support self-testable autonomic software. In: SAC '08. pp. 1664–1669. ACM, New York, NY, USA (2008)
12. Liu, H., Parashar, M., Hariri, S.: A component-based programming model for autonomic applications. In: ICAC '04. pp. 10–17. IEEE, Los Alamitos, CA, USA (2004)
13. Ma, Y.S., Kwon, Y.R., Offutt, J.: Mu Java 3 (November 2008), http://cs.gmu.edu/~offutt/mujava/ (June 2011)
14. Patouni, E., Alonistioti, N.: A framework for the deployment of self-managing and self-configuring components in autonomic environments. In: WOWMOM '06. pp. 480–484. IEEE Computer Society, Washington, DC, USA (2006)
15. Ramirez, A., Morales, B., King, T.M.: A self-testing autonomic job scheduler. In: ACM-SE 46. pp. 304–309. ACM Press, New York, NY, USA (2008)
16. van Renesse, R., Birman, K.P.: Autonomic Computing – A System-Wide Perspective / Editor(s): M. Parashar and S. Hariri. Taylor & Francis, Inc., Bristol, PA, USA (2007)
17. Rocha, C.R., Martins, E.: A strategy to improve component testability without source code. In: SOQUA/TECOS. pp. 47–62 (2004)
18. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
19. Skype Limited: Skype API (February 2007), https://developer.skype.com/ (June 2011)
20. Stevens, R., Parsons, B., King, T.M.: A self-testing autonomic container. In: ACM-SE 45. pp. 1–6. ACM Press, New York, NY, USA (2007)
21. The Eclipse Foundation: Test and Performance Tools Platform (Nov 2001), http://www.eclipse.org/tptp/ (June 2011)
22. Walls, C., Breidenbach, R.: Spring in Action. Manning Publications Co., Greenwich, CT, USA (2005)
23. Zhang, C., Sadjadi, S.M., Sun, W., Rangaswami, R., Deng, Y.: A user-centric network communication broker for multimedia collaborative computing pp. 1–5 (November 2006)
24. Zhang, J., Cheng, B.H.C., Yang, Z., McKinley, P.K.: Enabling safe dynamic component-based software adaptation. In: WADS. pp. 194–211 (2004)
25. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: AOSD '09. pp. 161–172. ACM, New York, NY, USA (2009)
26. Zhao, Y., Kardos, M., Oberthür, S., Rammig, F.J.: Comprehensive verification framework for dependability of self-optimizing systems. In: ATVA '05. Lecture Notes in Computer Science, vol. 3707, pp. 39–53. Springer (2005)