

A Comparative Case Study on the Engineering of Self-Testable Autonomic Software

Tariq M. King
Department of Computer Science
North Dakota State University
Fargo, ND 58108, USA
E-mail: tariq.king@ndsu.edu

Andrew A. Allen, Yali Wu and Peter J. Clarke
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
E-mail: {aalle004, ywu001, clarkep}@cis.fiu.edu

Alain E. Ramirez
IT Development
TracFone Wireless, Inc.
Miami, FL 33178, USA
E-mail: aesteva@tracfone.com

Abstract—A survey on the landscape of self-adaptive systems identified testing and assurance as one of the most neglected areas in the engineering of autonomic software. However, since the structure and behavior of autonomic software can vary during its execution, runtime testing is critical to ensure that faults are not introduced into the system as a result of dynamic adaptation. Some researchers have developed approaches and supporting designs for integrating runtime testing into the workflow of autonomic software. In this paper, we describe a comparative case study performed on three autonomic applications that were engineered to include an implicit self-test characteristic. The findings of our study provide evidentiary insight into the benefits and software engineering challenges associated with developing these types of systems.

Keywords—software testing; self-testing; autonomic computing; adaptive systems; case study

I. INTRODUCTION

The autonomic computing (AC) paradigm describes a new generation of software that is capable of self-configuration, self-optimization, self-protection, and self-healing [1]. AC seeks to address the problem of managing highly complex systems through the development of software that automates low-level decisions and tasks, and allows administrators to define system goals as high-level objectives [1], [2].

Autonomic software is typically characterized by dynamic adaptation, a self-management process in which the system adds, removes, and replaces its own components at runtime. In traditional software, such structural and behavioral changes would normally be implemented off-line during maintenance to repair faults, add new functionality, or adapt the system to a changing environment. At the end of maintenance, the modified software would then be retested to: (1) validate added or updated software features, and (2) ensure that new errors were not introduced into previously tested components, i.e., regression testing [3].

Although AC proposes to automate traditional software maintenance tasks using a highly dynamic self-management infrastructure, its architectural blueprint does not provide support for testing adaptive changes to the software [2]. Lack of runtime testing support in autonomic software can have a negative impact on its reliability, since faults may be introduced into the system as a result of dynamic adaptation. Researchers have therefore developed approaches for

integrating runtime testing into the workflow of autonomic software [4], [5].

King et al. [4] introduced an implicit *self-test* characteristic into autonomic software. Their approach tailors the self-management infrastructure of autonomic systems for software testing activities. Specialized autonomic test managers are deployed within the system to validate runtime changes to the software. Two validation strategies were formulated as part of their self-testing approach [4]: *Replication with Validation* (RV) – tests adaptive changes using copies of the managed resources of the system; and *Safe Adaptation with Validation* (SAV) – tests adaptive changes in-place, directly on the managed resources of the system.

Autonomic self-testing is supported by a reusable object-oriented design for developing prototypes of self-testable autonomic software [6]. Stevens et al. [7] demonstrated the feasibility of the RV strategy using an *autonomic container*, a data structure with self-configuring capabilities. Ramirez et al. [8] investigated deeper issues associated with RV by applying the autonomic container to the problem of short-term job scheduling. To the best of our knowledge, prior to the writing of this paper there were no prototypes that implement the SAV strategy.

Allen et al. [9] leveraged autonomic computing in a policy-driven approach for selecting and configuring communication services in a runtime environment, referred to as *Communication Virtual Machine* (CVM). CVM provides a model-driven platform for realizing user-centric communication services, and has been applied to the healthcare domain. The autonomic framework for CVM was developed based on elements of the design by King et al. [6], in order to make it self-test ready.

In this paper we extend the CVM with self-testing features according to the SAV strategy. The results of developing the self-testing CVM are presented as part of a comparative case study on engineering self-testable autonomic software. Our primary reason for conducting the study is to gain evidentiary insights on the benefits, and software engineering challenges associated with building these types of systems. Major contributions of this work are the experimental results and evaluation, which include static and dynamic analyses of three self-testing autonomic systems [7], [8], [9].

The rest of this paper is organized as follows: Section II contains background material. Section III provides a description of the applications used in the study. Section IV describes the experiments and setup environment. Section V contains the experimental results. Section VI evaluates the findings of the study and discusses the major observations, lessons learned, and threats to validity. Section VII is the related work, and in Section VIII we conclude the paper.

II. BACKGROUND

This section contains background material on autonomic computing, software testing, and autonomic self-testing.

A. Autonomic Computing (AC)

Autonomic computing (AC) refers to an embodiment of automated management features including self-configuration, self-protection, self-optimization, and self-healing in software-based systems [1]. Central to autonomic computing are the concepts of *autonomic managers*, and *managed resources*. Autonomic managers (AMs) consist of monitor, analyze, plan, execute (MAPE) functions that govern the resources of the system [1], [2]. The monitor function polls managed resources to observe their state, while the analyze function detects undesirable state conditions. In the presence of such conditions, the plan function formulates a remedy that is implemented by the execute function.

There are two categories of autonomic managers [2]: *Touchpoint AMs* – govern managed resources directly through their sensor and effector interfaces (called *touchpoints*); and *Orchestrating AMs* – manage one or more Touchpoint AMs. A manual management interface allows for human administration of the system. The complete set of architectural layers in an autonomic system are as follows (from top to bottom) [2]: *Manual Manager*, *Orchestrating AMs*, *Touchpoint AMs*, *Touchpoints*, and *Managed Resources*. A vertical layer of *Knowledge Sources* spans the management layers to allow information exchange.

B. Software Testing

Software testing is a dynamic process in which a finite set of test cases are executed on a software system [3]. Upon completion of a testing process, the actual results of test execution are compared with the expected results to determine if testing passed or failed.

Test criteria may be based on the program specification, implementation, or a combination of the two [3], [10]. During *specification-based testing*, the requirements drive the testing process by providing a means of checking the correctness of output, and measuring test adequacy. On the other hand, *implementation-based testing* determines adequacy by how thoroughly the program under test has been exercised. The effectiveness of implementation-based testing is generally stated in terms of coverage of the program's structure, e.g., all statements, all branches.

The fault detecting ability of a test set can be assessed using a technique known as *mutation testing* [10]. Mutation testing involves generating a set of programs, called *mutants*, that differ from the original program 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 [10]. For each mutant, if the test results differ from the original program on at least one test case, the mutant is said to have been *killed*, otherwise the mutant *lives*.

C. Autonomic Self-Testing

King et al. [11] introduced an implicit self-testing characteristic into autonomic software. Their approach tailors the existing self-management infrastructure for software testing activities such as test execution, code coverage analysis, and post-test evaluation. Self-management and testing concerns are kept separate by using independent test managers (TMs) that validate the adaptive actions of autonomic managers (AMs). To facilitate seamless integration into autonomic software, TMs have the same structure as AMs. However, the MAPE functions of TMs are specialized for testing activities via an interface to automated testing tools.

Under autonomic self-testing, TMs can operate according to two distinct validation strategies: *Replication with Validation* (RV) – tests adaptive changes using copies of the managed resources, maintained solely for testing purposes; and *Safe Adaptation with Validation* (SAV) – tests adaptive changes in-place, directly on the managed resources of the system. A reusable object-oriented design [6] was created to support the development of autonomic software systems that incorporate the self-testing characteristic.

III. DESCRIPTION OF APPLICATIONS

The following three Java applications were used in the study: Autonomic Container [7], Autonomic Job Scheduler [8], and Communication Virtual Machine [9]. In this section we provide details on the design of these applications, and describe their autonomic and self-testing features.

A. Autonomic Container (ACT)

Stevens et al. [7] introduced the concept of an *autonomic container* (ACT) to demonstrate the idea of self-testing in autonomic software. ACT is a data structure that possesses one or more of the self-management characteristics of autonomic computing [1]. The purpose of the autonomic container was to provide a simplistic, lightweight (i.e., small size and low complexity) implementation model of how autonomic software operates.

1) *Autonomic Features*: ACT incorporates the autonomic feature of self-configuration. Figure 1 provides a static view of the application. The managed resource of ACT is a stack (top-left) that can be dynamically reconfigured. A public interface, labeled `StackService`, provides data storage services to users including `push`, `pop`, `top`, `isEmpty`,

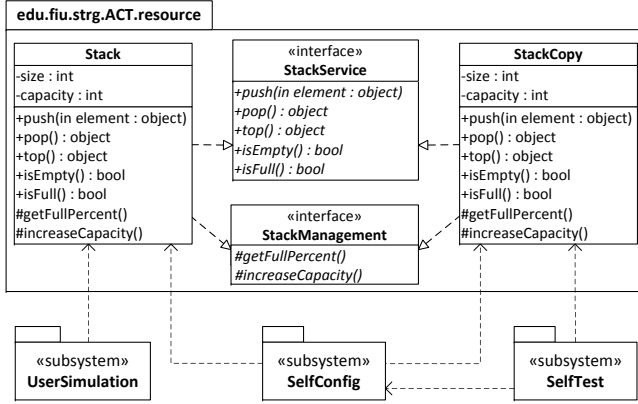


Figure 1. Design of the Autonomic Container (ACT)

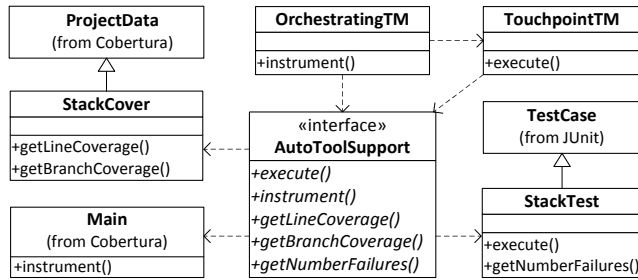


Figure 2. Design of the Self-Test Subsystem of ACT

and `isFull`. At runtime, when the stack reaches 80% of its full capacity, the subsystem labeled `SelfConfig` reconfigures the stack to hold more elements through the protected interface `StackManagement`.

Clients of the stack service are simulated via the `UserSimulation` subsystem, which can make calls to the stack in two modes: *Random* and *AlwaysPush*. Random mode determines whether to push or pop data elements by generating arbitrary boolean values, while the *AlwaysPush* mode continuously adds data elements to the stack. In both modes, the values of the data elements for push operations are random `java.lang.Integer` objects.

2) *Self-Test Features*: Self-testing in ACT is implemented according to the Replication with Validation (RV) strategy. Before accepting a self-configuration change request, the `SelfTest` subsystem of ACT executes a set of baseline regression tests on `StackCopy` using the newly specified structure. Figure 2 shows the design of the `SelfTest` subsystem of the ACT prototype. The `AutoToolSupport` interface provides services instrumenting the stack copy, executing regression tests, and collecting the test results. A suite of 15 test cases are stored in the class labeled `StackTest`, which extends `TestCase` from JUnit [12]. Branch and line coverage are computed via the `ProjectData` class of Cobertura [13]. To set

up validation, the `OrchestratingTM` instruments the `StackCopy` class in Figure 1 using `Main` from Cobertura. The `TouchpointTM` then invokes `StackTest` to validate structural changes to the stack.

B. Autonomic Job Scheduler (AJS)

Ramirez et al. [8] applied the autonomic container in the context of a more complex problem, *short-term job scheduling*. In an operating system, a short-term scheduler determines the order in which jobs in memory receive the attention of the processor. Many job scheduling algorithms have been developed to address the various performance criteria, e.g., response time, throughput.

AJS is an application that simulates the behavior and environment of a self-managing, short-term job scheduler for an operating system. A pool of agents, representing the processors of the system, service jobs according to a high-level scheduling algorithm. AJS assumes that the system has two distinct, goal-oriented modes: *Interactive* – jobs in the request queue have been submitted by on-line users who are awaiting a response; and *Batch* – jobs in the request queue have been collected over a period of time and do not require an immediate response.

1) *Autonomic Features*: AJS includes self-configuration and self-optimization features. Job requests are stored in a queue container which structurally reconfigures its capacity in a similar fashion to the autonomic container. Behavioral changes are realized through dynamic replacement of the scheduling algorithm. Two scheduling algorithms were implemented: *Shortest Request Next* and *First-Come First-Serve*. Self-optimization is achieved by adjusting the number of processing agents in proportion to the workload. A subsequent point release of AJS (v1.1) incorporates self-protection, by temporarily blocking users who repeatedly cause the job queue to throw exceptions.

Figure 3 provides a static view of AJS. The self-management infrastructure of the application is comprised of: (1) two managed resources for the job request container and the processing agents, `RequestContainer` and `RequestAgents` respectively; (2) a `ContainerCopy` and an `AgentsCopy` for use when testing autonomic changes to the managed resources; (3) two touchpoint AMs for self-configuration and self-optimization, `RCSelfConfig` and `RASelfOptim` respectively; (4) an orchestrating AM for coordinating self-management, labeled `OAMScheduler`; and (5) an external knowledge source labeled `KSSScheduler`.

2) *Self-Test Features*: Self-testing in both versions of AJS follow the RV strategy, and is supported by JUnit [12] for test execution, and Cobertura [13] for calculating line and branch coverage. As shown in Figure 3, the self-testing infrastructure of AJS is comprised of: (1) two touchpoint TMs for validating changes to each managed resource, labeled `RCSelfTest` and `RASelfTest` respectively; and

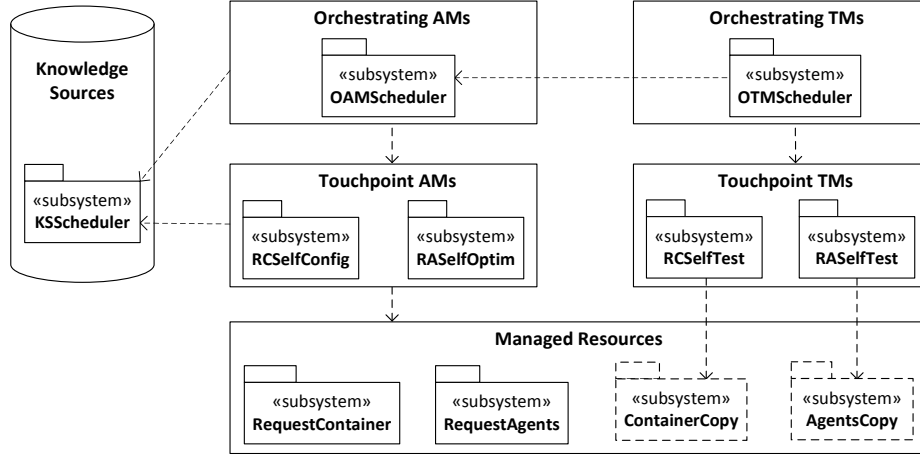


Figure 3. Design of the Self-Testing Autonomic Job Scheduler (AJS)

(2) an orchestrating TM to coordinate the testing process, labeled `OTMScheduler`. The test set for the initial version of AJS consisted of a total of 32 test cases. An additional 7 test cases were developed for AJS v1.1.

C. Communication Virtual Machine (CVM)

Deng et al. [14] present a model-driven platform for realizing user-centric communication services, referred to as *Communication Virtual Machine (CVM)*. CVM is the result of a collaboration between FIU SCIS and Miami Children’s Hospital, where the application domain is the realization of healthcare communication services.

CVM separates the concerns of communication modeling, synthesis, coordination, and the actual delivery of the communication services into self-contained layers [14], [15]. These layers are: *User Communication Interface* – a modeling environment for users to specify their communication requirements as schemas; *Synthesis Engine* – a suite of algorithms for transforming user communication schemas into an executable control script; *User-Centric Communication Middleware* – a handler that executes control scripts to coordinate the delivery of communication services, independent of the network configuration; and *Network Communication Broker (NCB)* – a uniform API that configures network devices and protocols to realize communication.

1) *Autonomic Features*: Allen et al. [9] leverage autonomic computing, and open-platform communication APIs, in the provision of a comprehensive set of communication services for CVM. Figure 4 shows the design of the NCB layer of CVM. Through a series of dynamic adaptations, NCB self-configures to use multiple communication frameworks such as Skype [16], Smack [17], and Native NCB [15] (bottom-right). An NCB management layer exposes the high-level communication services to its upper layer via a network independent API (top of Figure 4).

When a request for communication is received, it is stored in the knowledge source `KSCommunication` (left) via the `CallQueuing` subsystem. A communication services manager (CSM) monitors the call queue for new requests and invokes the interface `NCBBridge` (bottom-right). The `NCBBridge` facilitates access to network-level communication frameworks, which have been made available through open-platform APIs such as Skype [16], Smack [17], and Native NCB [15].

The communication services provided to the CSM via `NCBBridge` are: creating and destroying communication connections; adding and removing participants from a connection; and enabling and disabling different types of communication media in a connection. The adapter design pattern was used to standardize the APIs of each open-platform communication framework. In Figure 4, this is represented by the interfaces labeled `SkypeAdapter` and `SmackAdapter` in the managed resource layer. Using the adapters allows the NCB to be extensible for inclusion of other communication frameworks, as indicated by the dotted package `NewFramework`.

A touchpoint AM, labeled `TAMCommunication`, directly manages the communication frameworks and is responsible for adapting the underlying network configuration. An orchestrating AM, labeled `OAMCommunication`, monitors the CSM and touchpoint AM, and also analyzes the requests in the call queue to determine if self-configuration is required. Requests for NCB self-management have two general forms: (1) a user’s explicit request for communication that is not supported by the current configuration, thereby requiring self-configuration; and (2) the occurrence of other external events such as failure in a framework, thereby requiring self-healing.

2) *Self-Test Features*: Self-testing in the NCB adheres to the Safe Adaptation with Validation (SAV) strategy.

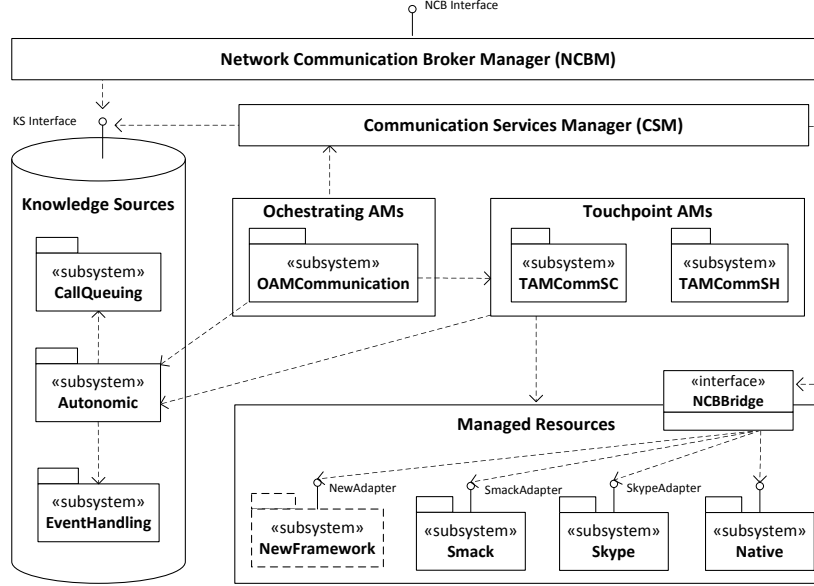


Figure 4. Design of the Autonomic Network Communication Broker (NCB) Layer of CVM

An orchestrating TM was implemented to monitor the orchestrating AM labeled *OAMCommunication* for the generation of change requests. These change requests were then validated in-place by a touchpoint TM, after disabling caller functions in the CSM and touchpoint AMs to ensure safety. The automated tools JUnit [12] and Cobertura [13] were used to support self-testing in the NCB. A total of 41 tests were developed to validate self-adaptations in the NCB.

D. Summary

Table I summarizes the autonomic and self-testing features of the applications used in the study. Each row of the table consists of the following information: application name; validation strategy employed (VS); test set size (TS); number of test managers (TM); number of self-testable autonomic components (ST); and indicators of the autonomic features of self-configuration (SC), self-optimization (SO), self-protection (SP), and self-healing (SH).

	Name	VS	Self-Testing			Self-Management			
			TS	TM	ST	SC	SO	SP	SH
1	ACT	RV	15	2	1	√			
2	AJS 1.0	RV	32	3	2	√	√		
3	AJS 1.1	RV	39	4	3	√	√	√	
4	CVM	SAV	41	2	1	√			√

Table I
DESCRIPTIVE SUMMARY OF APPLICATIONS USED IN THE CASE STUDY

IV. EXPERIMENTAL SETUP

This section describes the setup of the experiments that were used to evaluate autonomic self-testing [4], and its supporting design method [6]. Experiments are categorized according to the effectiveness of applying the approach with respect to: (1) developing the supporting infrastructures and prototypes of self-testable autonomic software, (2) minimizing negative impacts on system performance during self-testing; and (3) detecting faulty change requests and exercising the prototype implementations during self-testing.

A. Development Experiments

The first set of experiments involved assessing the development effort for the three applications, focusing on the autonomic and self-testing features. Infrastructures for self-management and automated testing tool support were developed in Java 1.5 using the Eclipse 3.2 SDK. The development teams used synchronized threads, blocking queues, generics, and reflection in Java to implement self-management features.

Size and other complexity metrics of the software artifacts produced during development were generated using the metrics plugin for Eclipse. The following metrics were automatically exported from Eclipse in XML format as an estimate of development overhead: lines of code, number of methods, number of classes, and cyclomatic complexity. A windows-based Intel(R) Core 2 Duo 2.4GHz PC with 4GB RAM was used to collect the development metrics.

B. Performance Experiments

The second set of experiments focused on evaluating the runtime performance of different variants of the AJS

Application	Mutated Resource	Mutant Categories	# Mutants
ACT	Stack	M1. Improper Capacity Increase	5
		R1. Random	7
AJS			
AJS	RequestContainer	Reused Mutants from M1.	5
	RequestAgents	M2. Improper Agent Enablement	3
	RequestAgents	M3. Improper Agent Disablement	3
	SchedAlgorithm	M4. Improper Job Ordering	2
	SchedAlgorithm	M5. Invalid Algorithm Syntax	1
		R2. Random	20
CVM			
CVM	NCBAdapters	M6. Improper Initialization / Reset	5
	NCBAdapters	M7. Improper Participant Identification	3
	NCBAdapters	M8. Improper Session Identification	3
	NCBAdapters	M9. Improper Media Re-Configuration	4
		R3. Random	24
Total Mutants			85

Table II
MUTANTS GENERATED TO SIMULATE FAULTY CHANGE REQUESTS

and CVM prototypes. More specifically, the performance experiments involved: (1) comparing a non-self-test variant of AJS with a self-test variant that uses a distributed testing process; and (2) comparing a self-test variant of CVM with a thread-enhanced self-test variant.

Experiment runs for the AJS involved simulating identical sets of user actions on both variants of the application to induce self-testing. However, for the distributed self-test variant, the testing infrastructure and process was located on a separate computational node than that of the core application. This was achieved by using Java Remote Method Invocation [18] library to communicate with the distributed self-test subsystem.

In the CVM performance experiments, a self-configuration from a Skype two-way AV call to a Smack three-way AV call was used to induce the self-testing process. However, self-testing in CVM was interleaved with the core application on the same node. A timeout of 250 milliseconds was introduced into the MAPE functions in an attempt to enhance the interleaving of the core application during self-testing.

Performance metrics associated with memory usage and thread utilization were collected using the Eclipse Test and Performance Tools Platform (TPTP) [19]. A detailed thread and timestamp analysis of the experimental runs were exported from TPTP into Comma Separated Value (CSV) format for comparison. The CVM self-test performance results were obtained by instrumenting the call initiator node

only. All machines involved in this round of experiments were windows-based, webcam-enabled, Intel(R) Pentium 4 PCs with 2GB RAM or greater.

C. Test Set Quality Experiments

The third set of experiments were aimed at measuring the effectiveness of test sets in revealing faults, and exercising program code. Mutation analysis was performed on all three applications by seeding artificial faults into the change requests for managed resources. This allowed us to simulate the generation of incorrect structural and behavioral changes to the software, induce self-testing, and record the results.

Table II describes the mutants that were created for the ACT, AJS, and CVM prototypes. Each row of the table provides the following information: abbreviated application name, managed resource involved in the mutation, categories for the mutants, and number of mutants in each category. The mutants in categories with the prefix **M-** were manually created by the testing team for the application, while those prefixed with **R-** were generated automatically using method-level mutation operators from MuJava 3 [20].

Runtime test sets for each adaptable managed resource were written in JUnit 3.8.1 [12]. Cobertura 1.9 [13] was used to instrument the managed resources during testing to calculate line and branch coverage. The XML reports generated by Cobertura were parsed to extract code coverage results. In cases where JUnit tests failed, the results for line and branch coverage were omitted from the computation. The

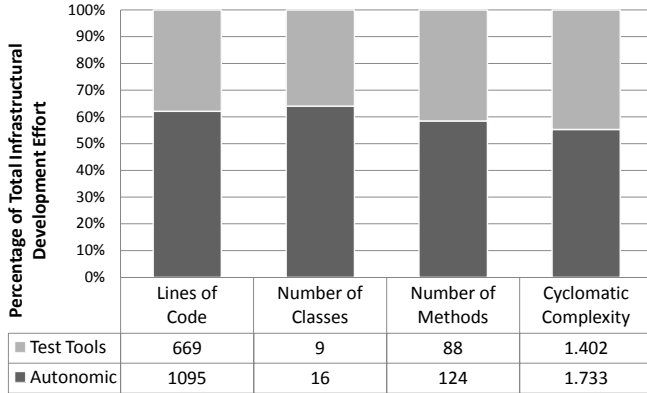


Figure 5. Comparison of Infrastructural Development Effort

following metrics were recorded in this set of experiments: number of test case failures, percentage of program lines and branches covered, and mutation scores. Test results for each prototype were gathered using a windows-based, Intel Pentium 4 PC with 2GB RAM or greater.

V. RESULTS

This section contains the findings of the different sets of experiments, and has been classified accordingly.

A. Development Results

The results of the development experiments are divided into infrastructure and prototype development.

1) *Infrastructure*: Figure 5 provides a summary of the results obtained from building the autonomic infrastructure and testing tool support. Each column of the table at the bottom of the figure provides measurements for the two infrastructures based on lines of code, number of classes, number of methods, and cyclomatic complexity. The stacked bar chart in Figure 5 compares the relative effort needed to develop each infrastructure. For example, as indicated by the leftmost bar in Figure 5, incorporating autonomic support accounted for $\sim 62\%$ of total infrastructural development effort with respect to lines of code. The remaining 38% was dedicated to providing testing support.

2) *Prototype*: Figures 6 and 7 present the development metrics collected from static analysis of the prototype implementations. The prototypes were compared at three different stages of their development. The key at bottom left of Figures 6 and 7 indicate these stages, which are: (1) *Non-Autonomic* – implementation before any autonomic features were incorporated, (2) *Autonomic* – implementation after incorporating autonomic features; and (3) *Autonomic-ST* – implementation after incorporating autonomic and self-test features. Only the lines of code (LOC) metric was used for prototype implementation comparisons.

Figure 6 presents a stacked bar chart for comparing the raw LOC data measurements of the ACT, AJS v1.0, and

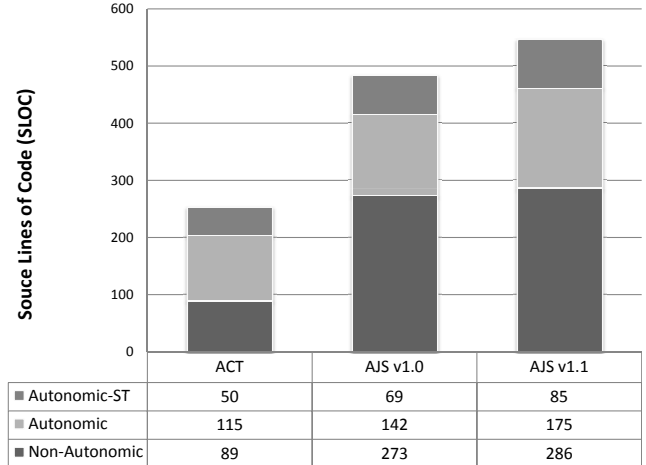


Figure 6. Comparison of Incremental ACT and AJS Development Effort

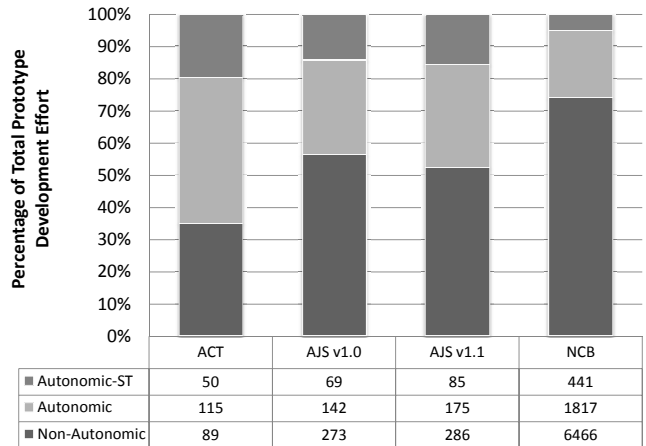


Figure 7. Comparison of Relative Development Effort for Prototypes

AJS v1.1 prototypes. These three prototypes were built incrementally, and therefore provide a means of analyzing the effect of applying the RV strategy as the application size increases. The stacked representation allows the prototypes to be compared both vertically within the same version, and horizontally across different releases.

The bar chart in Figure 7 shows the relative development effort of building different aspects of the prototypes, including the NCB layer of CVM. As indicated by the right-most table column at the bottom of Figure 7, the non-autonomic version of NCB is a medium-sized software component with a total of 6466 LOC. Introducing autonomic and self-testing capabilities into NCB increased its size by 1817 LOC and 441 LOC, respectively. Due to the large size of the NCB in comparison to the other prototypes, the y-axis of the bar chart in Figure 7 measures the percentage of total LOC, rather than absolute values.

B. Performance Results

The results of the performance experiments are divided into the findings for AJS and CVM.

1) *Autonomic Job Scheduler*: For the AJS performance experiment, the percentage difference between the non-self-test and distributed self-test variants were $< 1\%$ for both thread utilization and memory usage.

2) *Communication Virtual Machine*: The performance results for the NCB layer of CVM is shown in Figure 8. Five sample runs of the Skype two-way AV to Smack three-way AV call were performed as part of the experiment, and are shown along the x-axis of Figure 8. The y-axis of the figure shows the percentages of AM and TM thread utilization, which were recorded during each sample run.

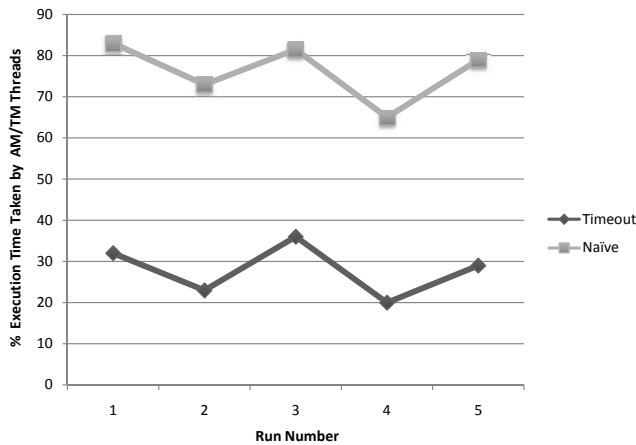


Figure 8. Thread Utilization of AMs and TMs in NCB layer of CVM

C. Test Set Quality Results

The results of the test set quality experiments are divided into mutation analysis and code coverage.

1) *Mutation Analysis*: We calculated the mutation scores for each application by taking the number of mutants killed in the application, and dividing it by the total number of mutants that were not equivalent to the original application. The mutation scores were as follows: for ACT we computed a score of 12/12 (100%), AJS was 32/34 (94.1%), and CVM was 35/39 (89.7%).

2) *Code Coverage*: The percentages of program code coverage for the applications were as follows: for ACT we recorded 80% line coverage and 75% branch coverage, AJS was 77% line and 70% branch, and CVM was 63% line and 57% branch.

VI. EVALUATION

The primary objective of conducting the case study is to gain evidentiary insights on the benefits, and software engineering challenges associated with developing self-testable autonomic software. Our preliminary investigation sought to seamlessly integrate self-testing into existing autonomic

software systems. However, when no open-source autonomic applications could be located, our focus shifted to building the ACT and AJS prototypes as a means of investigating the Replication with Validation strategy. Research collaborations with the CVM team [21], provided a real-world autonomic system for investigating Safe Adaptation with Validation.

This section describes the major observations, challenges, and lessons learned from conducting the comparative case study on ACT, AJS, and CVM. Threats to the validity of our experimental results are also discussed.

A. Observations

We now highlight various aspects of the experimental results to substantiate the use of autonomic self-testing. In particular, evidence to support the benefits during system development and execution is emphasized.

1) *System Development*: The results on infrastructural development (Figure 5) reveal that on average it took 10-20% percent less effort to develop self-test support than the autonomic infrastructure. This proved to be true for each of the size metrics used in the evaluation, as well as the cyclomatic complexity metric.

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

Analysis of the incremental prototype development in Figure 6 reveals additional evidence that the development overhead of self-testing was less than self-management. In all cases, even though there was a one-to-one ratio of AMs to TMs, there was significantly less code dedicated to implementing self-test features. This was due largely to heavy reuse of the autonomic infrastructure in implementing the self-test engine. Code modifications for linking TMs to the automated testing tool support were relatively simple.

Comparing the ACT, AJS, and CVM size metrics in Figure 7 shows that self-test drivers took 15-25% less development effort than those for autonomic management. The self-test policies for TMs were also relatively easy to develop, and at most required formulating one or two boolean state mapping for symptom detection. On the other hand, some AM policies implemented as much as seven state mappings depending on the system requirements.

2) *System Execution*: Performance results for AJS under distributed Replication with Validation showed that this strategy provides a highly transparent testing process. Less than an 1% increase in processing and memory usage was recorded, and there were no visible signs of degradation. This is because TMs monitored AMs remotely, and the cost

of periodic remote method invocations was low (i.e. simply checking the state of one or two boolean variables).

Introducing a small timeout into the AMs and TMs of the NCB significantly improved its runtime performance. As shown in Figure 8, without the timeout 65-85% of process interleaving was dedicated to AMs and TMs. Under such conditions there were high levels of degradation in the CVM with respect to establishing communication connections. Once the timeout in the MAPE was activated, the AMs and TMs only utilized 20-35% of the process execution time. This placed the degradation of the CVM within acceptable bounds. However, there was a 250 millisecond delay trade-off with respect to symptom detection.

B. Lessons Learned

Synchronization between components and ensuring harmony between the closed control loops were the major challenges experienced when developing the ACT and AJS prototypes. Although both of these applications are relatively small autonomic systems, the heavy use of threads and lack of safety mechanisms made debugging them quite challenging. Feedback from the ACT and AJS prototypes led us to update the initial manager design with safety mechanisms. In general, tests to validate the adaptive changes within ACT and AJS were not very difficult to develop.

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.

Many of the command line testing tools had poor error reporting capabilities, and the slightest deviation from their command line parameters would result in failure traces that were difficult to understand. As a result, special care had to be taken to ensure that all class path configurations were included, and ordered correctly within the automated testing tool support implementation.

C. Threats to Validity

A major threat to the validity of the results is the inexperience of the autonomic development and testing teams.

With the exception of the CVM team, many of the software engineers involved in building the prototypes were relatively new to the problem domain and field of AC. Furthermore, the lack of open-source AC projects meant that the developers did not have access to other detailed designs and implementations to guide their activities. The information provided by such artifacts would have been highly valuable during the specification of AMs, managed resources, and knowledge sources.

Another possible threat to validity is not considering the effort required to develop autonomic and self-test features irrespective of the size of the source code, and automated testing scripts. For example, although the lines of code metric for the test cases and test drivers for the CVM prototype was not very high, making the NCB self-testable took a significant amount of time due to its complexity. However, this may have been the result of the inexperience of the CVM team in developing test scenarios for a real-time, communication-intensive application.

VII. RELATED WORK

Testing and assurance in autonomic software has been a highly neglected research topic, with very few works being cited in the literature [22]. To the best of our knowledge, this is the first comparative case study to be performed on integrating runtime testing into autonomic software.

The work by Da Costa et al. [5] is most closely related to our work. It presents the Java Self-Adaptive Agent Framework + Test (JAAF+T). JAAF+T incorporates runtime validation into autonomic agents by directly modifying their MAPE control loops to include a test activity [5]. Autonomic agents have combined responsibilities of adaptation and testing. The feasibility of JAAF+T is demonstrated by a case study that implements the test activity in an adaptive system for generating susceptibility maps.

Similar to the CVM prototype presented in this paper, the JAAF+T susceptibility maps application applies autonomic computing to a real-world problem. Including this application in our study could enhance our evaluation by providing: (1) additional static and dynamic analysis results from performing runtime testing on another real-world AC application; and (2) an alternative self-test model to allow comparison of two different approaches for integrating runtime testing into autonomic software.

Munoz and Baudry [23] present an approach to testing the adaptation logic of self-adaptive systems called *Artificial Shaking Table Testing* (ASTT). Using ASTT the authors are able to select test data that simulates complex environmental variations called *context-shakes*. The synthesis of context-shakes is modeled as a search problem. An experimental evaluation of ASTT suggests that it is an effective approach, for detecting faults associated with the correctness and completeness of adaptation policies.

The work on ASTT tackles the problem of whether or not an adaptive system is responding correctly to changes in its environment. Munoz and Baudry [23] therefore address external aspects of testing adaptive systems. On the other hand, we address a slightly different but closely related problem: “If an adaptive change is applied to the system at runtime, will errors be introduced into the software?”. Our viewpoint is focused on how adaptation will affect the internal structure and behavior of system components. The two approaches are therefore highly complementary, and both research directions are necessary to ensure the overall correctness of autonomic and self-adaptive software.

VIII. CONCLUSION

Conducting the case study revealed many practical issues surrounding the engineering of self-testable autonomic software, including: (1) the benefits to be gained from having reusable designs, and proper tools and frameworks for building self-testable autonomic software; and (2) the challenges associated with validating adaptive changes to autonomic software at runtime. Future work calls for investigating dynamic test generation, planning, and scheduling approaches in autonomic software; and performing further case studies in these areas.

IX. ACKNOWLEDGEMENTS

The authors would like to thank Ronald Stevens Jr., Barbara Morales-Quinones, and Rodolfo Cruz for their contributions to this work. This work was supported in part by the National Science Foundation (NSF) under grants IIS-0552555 and HRD-0833093. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or TracFone Wireless, Inc.

REFERENCES

- [1] J. Kephart and D. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–52, January 2003.
- [2] IBM Autonomic Computing Architecture Team, “An architectural blueprint for autonomic computing,” IBM, Hawthorne, NY, Tech. Rep., June 2006.
- [3] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.
- [4] T. M. King, A. E. Ramirez, R. Cruz, and P. J. Clarke, “An integrated self-testing framework for autonomic computing systems,” *JCP*, vol. 2, no. 9, pp. 37–49, 2007.
- [5] A. D. da Costa, C. Nunes, V. T. da Silva, B. Fonseca, and C. J. P. de Lucena, “JAAF+T: a framework to implement self-adaptive agents that apply self-test,” in *SAC '10*. New York, NY, USA: ACM, 2010, pp. 928–935.
- [6] T. M. King, A. Ramirez, P. J. Clarke, and B. Quinones-Morales, “A reusable object-oriented design to support self-testable autonomic software,” in *SAC '08*. New York, NY, USA: ACM, 2008, pp. 1664–1669.
- [7] R. Stevens, B. Parsons, and T. M. King, “A self-testing autonomic container,” in *ACM-SE 45*. New York, NY, USA: ACM Press, 2007, pp. 1–6.
- [8] A. Ramirez, B. Morales, and T. M. King, “A self-testing autonomic job scheduler,” in *ACM-SE 46*. New York, NY, USA: ACM Press, 2008, pp. 304–309.
- [9] A. A. Allen, Y. Wu, P. J. Clarke, T. M. King, and Y. Deng, “An autonomic framework for user-centric communication services,” in *CASCON '09*. New York, NY, USA: ACM, 2009, pp. 203–215.
- [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] T. M. King, D. Babich, J. Alava, R. Stevens, and P. J. Clarke, “Towards self-testing in autonomic computing systems,” in *ISADS '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 51–58.
- [12] E. Gamma and K. Beck, “JUnit 3.8.1,” 2005, <http://www.junit.org/index.htm> (February 2011).
- [13] M. Doliner, G. Lukasik, and J. Thomerson, “Cobertura 1.9,” 2002, <http://cobertura.sourceforge.net/> (February 2011).
- [14] Y. Deng, S. M. Sadjadi, P. J. Clarke, V. Hristidis, R. Rangaswami, and Y. Wang, “CVM-a communication virtual machine,” *J. Syst. Softw.*, vol. 81, no. 10, pp. 1640–1662, 2008.
- [15] C. Zhang, S. M. Sadjadi, W. Sun, R. Rangaswami, and Y. Deng, “A user-centric network communication broker for multimedia collaborative computing,” *CollaborateCom '06*, p. 28, 2006.
- [16] Skype Limited, “Skype API,” February 2007, <https://developer.skype.com/> (February 2011).
- [17] Jive Software, “Smack API,” November 2008, <http://www.igniterealtime.org/projects/smack/> (February 2011).
- [18] Sun Microsystems, Inc., “Trail: Java Remote Method Invocation (RMI),” February 2008, <http://java.sun.com/docs/books/tutorial/rmi/index.html> (February 2011).
- [19] The Eclipse Foundation, “Test and Performance Tools Platform,” Nov. 2001, <http://www.eclipse.org/tptp/> (February 2011).
- [20] Y. S. Ma, Y. R. Kwon, and J. Offutt, “Mu Java 3,” November 2008, <http://cs.gmu.edu/~offutt/mujava/> (February 2011).
- [21] P. J. Clarke, Z. Yang, R. France, T. M. King, F. M. Costa, A. A. Allen, and Y. Wu, “Modeling and Realizing User-Centric Communication Services,” 2010, <http://cvm.cis.fiu.edu/> (February 2011).
- [22] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.
- [23] F. Munoz and B. Baudry, “Artificial table testing dynamically adaptive systems,” INRIA, Research Report RR-6866, 2009. [Online]. Available: <http://hal.inria.fr/inria-00365874/en/>