A Self-Testing Autonomic Job Scheduler

Alain E. Ramirez
School of Computing and Information Sciences
Florida International University
Miami, FL 33199
aeste010@cis.fiu.edu

Barbara Morales
Department of Computer Engineering
University of Puerto Rico
Mayaguez, PR 00681-9000
bmq21964@uprm.edu

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

ABSTRACT

Although researchers have been exchanging ideas on the design and development of autonomic systems, there has been little emphasis on validation. In an effort to stimulate interest in the area of testing these self-managing systems, some researchers have developed lightweight prototypical applications to show the feasibility of dynamically validating runtime changes to autonomic systems. However, in order to reveal some of the greater challenges associated with building dependable autonomic systems, more complex prototype implementations must be developed and studied. In this paper we present implementation details of a self-testable autonomic job scheduling system, which was used as the basis for our investigation on testing autonomic systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification— *validation, reliability*; D.2.10 [**Software Engineering**]: Design—*Object-Oriented Design Methods*

General Terms

Verification, Reliability, Design.

Keywords

Autonomic Computing, Self-Testing, Job Scheduling.

1. INTRODUCTION

Although there are many challenges associated with the runtime validation and verification of autonomic computing systems [6, 7], very few researchers and practitioners have addressed these important software engineering issues. Autonomic systems can dynamically configure, optimize, protect and heal themselves [4] in response to changing environmental conditions. These types of systems incorporate self-management facilities with the aim of reducing the burden of manually managing complex computing systems [6]. However, the inclusion of self-management capabilities means that autonomic systems are generally characterized by complex runtime behavior, thereby requiring online testing mechanisms to validate runtime changes to system components.

Some researchers have been investigating the use of an integrated self-testing framework to validate dynamic changes and re-configurations in autonomic computing systems [8, 9, 12]. Change requests to managed resources are intercepted by an online testing framework to ensure that: (1) new errors will not be introduced into previously tested components; and (2) the behavior of newly added or adapted components is validated prior to their use in the system [9]. The idea is to prevent change requests that will have an adverse effect on the autonomic system from being implemented.

One proposed strategy for testing autonomic systems at runtime is to maintain copies of managed resources soley for testing purposes, known as *replication with validation* [9]. Change requests are implemented on these resource copies and validated prior to be incorporated into the actual managed resources of the autonomic system. Stevens et. al [12] presented a prototype of a self-testing autonomic container to show the feasibility of the replication with validation strategy. The prototype provided online testing services to a stack of random numbers that could automatically reconfigure its size at runtime [12]. However, such a lightweight application is limited with respect to its ability to reveal the intricacies of dynamically validating autonomic systems.

As research continues in the area of testing autonomic systems, it is important to study more detailed prototype designs and implementations. Such works prove highly valuable with respect to providing concrete examples of the proposed techniques for constructing dependable autonomic systems. In this paper we extend the work in [12] by investigating the replication with validation strategy in the context of the problem of job scheduling. The main contributions of this work are that it:

1. Elaborates on the detailed design and implementation of a self-testable autonomic job scheduling system, *Autonomic Job Scheduler*.
2. Defines explicit dependency relationships between the self-testing framework and the autonomic computing system of the replication with validation strategy.
3. Discusses the major challenges and lessons learned from developing a prototype of a self-testable autonomic system.

ACM-SE '08, March 28–29, 2008, Auburn, AL, USA.

This paper is organized as follows: the next section contains information on autonomic computing. Section 3 provides background information on testing autonomic computing systems. Section 4 introduces the concept of an autonomic job scheduler. Section 5 elaborates on the implementation details of the prototype. Section 6 presents related work, and in Section 7 we conclude and discuss future work.

2. AUTONOMIC COMPUTING

Autonomic Computing (AC) applies self-management concepts such as those found in the human biological system to computing systems. AC is based on the autonomic nervous system and its seamless ability to free the brain from conscious involvement in vital functions such as breathing, homeostasis, and blood pressure regulation [5]. The autonomic nervous system monitors changes in the human body and adjusts various functions in response to stimuli to maintain balance in the system. In 2001, IBM proposed that computing systems be built in a similar manner that facilitates self-management, thereby freeing administrators from the tedious tasks associated with managing highly complex computing systems [6].

The main characteristics of autonomic computing systems are: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection* [4, 6, 10]. Self-configuration involves the automated installation, configuration and integration of system components. Self-optimization monitors and controls the utilization of computing resources based on factors such as memory usage, processing, and throughput. Self-healing detects, diagnoses, and repairs localized software and hardware problems automatically. Self-protection seeks to defend the system against malicious attacks, potential threats or cascading failures [10].

Self-management behavior in autonomic computing systems is realized through closed control loops [4] within entities called *autonomic managers*. Autonomic managers implement *monitor*, *analyze*, *plan*, and *execute* (MAPE) functions on the parts of the system which require automated management [4]. The monitor collects state information from a 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 formulated for execution on the resource. AMs also contain a knowledge component which allows access to data shared by the MAPE functions.

Figure 1 shows the layered architecture of an autonomic computing system which is composed of *managed resources*, *touchpoints*, *touchpoint autonomic managers*, *orchestrating autonomic managers*, *a manual manager*, and *knowledge sources* [4]. Managed resources include any entities of the system for which autonomic management services are being provided. Touchpoints implement the sensor and effector interfaces of the managed resources to facilitate state monitoring and adjustment functions, respectively. Touchpoint AMs interact directly with managed resources through their touchpoint interfaces to achieve low-level management. At a higher level, orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for individual resources [4]. The manual manager facilitates the activities of the human administrator through the provision of a management console. A vertical layer of knowledge sources implements repositories that can be used to extend the capabilities of AMs, and may be updated via the manual manager [4].

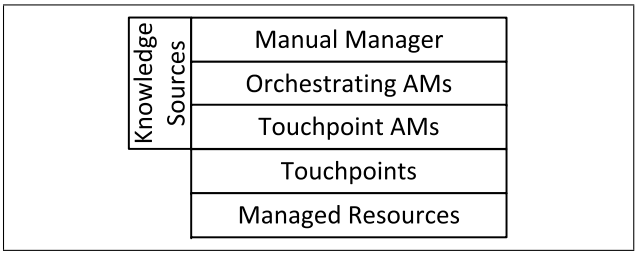

Figure 1: Architecture for Autonomic Computing.

3. TESTING AUTONOMIC SYSTEMS

King et al. [9] presented an integrated self-testing framework for autonomic computing systems. The approach introduced self-testing as an implicit characteristic of autonomic systems regardless of the self-management features being implemented. Two general strategies were proposed for validating runtime changes in autonomic systems – *safe adaptation with validation* and *replication with validation* [9]. In this section, we provide background information on the self-testing framework for autonomic systems in [9], and overview the replication with validation strategy.

3.1 Self-Testing Framework

The integrated self-testing framework is composed of test managers (TMs), test knowledge sources, and auxiliary test services that collaborate to provide validation services to the autonomic system [9]. Like autonomic managers, TMs may also be Orchestrating or Touchpoint. Orchestrating TMs interface with the autonomic software system, and coordinate the high-level testing activity by managing Touchpoint TMs. Touchpoint TMs perform low-level testing tasks directly on managed resources using the touchpoint layer of the autonomic system [8]. These low-level testing capabilities include the ability to control the internal state of the managed resource and observe its output. To perform their testing duties, TMs are composed of components that implement control loops that are consistent with the MAPE structure of AMs [9].

Test knowledge sources contain artifacts such as validation policies, test cases, test logs, and test histories, which are shared between Orchestrating TMs and Touchpoint TMs [9]. Similar to the knowledge sources component of the autonomic system, test knowledge sources may implement repositories or registries to extend the capabilities of TMs and facilitate automation. For example, administrators could register new testing strategies or new validation policies via the test knowledge sources. TMs would then monitor the test knowledge sources component for changes and automatically update their own built-in knowledge with the new test information. The auxiliary test services component provides Orchestrating TMs with access to external or third-party automated support tools; and implements facilities for manual test management [9].

3.2 Replication with Validation

Under the replication with validation strategy change requests for managed resources are first implemented on a readily available copy of the resource, and testing is performed using the copy [9]. The autonomic system must therefore maintain copies of managed resources solely for validation purposes. The steps of the replication with validation strategy are summarized as follows [9]:

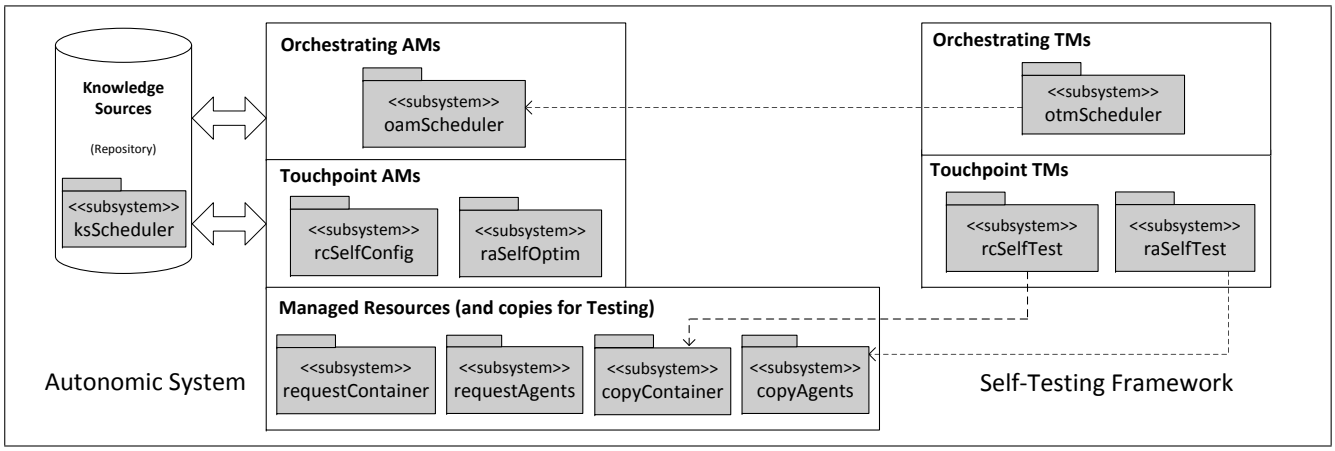

Figure 2: High-Level Architectural Model for a Self-Testing Autonomic Job Scheduler.

1. A Touchpoint AM detects that the managed resource is ready for self-management and generates a change request, which is intercepted by an Orchestrating AM.
2. The Orchestrating AM notifies the testing framework that validation is required, and initiates the replication with validation strategy.
3. The Orchestrating and Touchpoint TMs of the testing framework perform validation on a copy of the managed resource with the change request implemented.
4. If validation passes, the autonomic system is notified to accept the change; otherwise the change request is rejected.

In some cases, the test evaluation at (4.) may fail due to inadequate test coverage and therefore, depending on the validation policy, testing may be continued in an attempt to acquire the desired level of test coverage.

4. AUTONOMIC JOB SCHEDULER

Testing autonomic systems at runtime poses many challenges for researchers and practictioners in the field of autonomic computing [7, 6]. Stevens et. al [12] attacked the problem by investigating a lightweight, simplified application with self-management characteristics called an autonomic container. They integrated the notion of self-testing into the autonomic container using the replication with validation strategy by implementing a self-configurable stack of random numbers.

This paper extends the work by Stevens et. al [12] by applying the replication with validation strategy in the context of a more realistic problem, which we refer to as an *Autonomic Job Scheduler* (AJS). In this section, we discuss the main characteristics and autonomic features of AJS , and present its high-level architectural model with integrated self-testing capabilities.

4.1 Characteristics and Features

Job scheduling in a computing system is generally performed at three levels [11]: (1) *long-term* - determines which jobs are admitted to the system for processing; (2) *short-term* - determines which job in memory receives the attention of the processor; and (3) *medium-term* - determines which processes gets swapped in and out of the system temporarily to free up memory. AJS is a short-term scheduler that simulates activities such as job submission, servicing and monitoring of job executions, while incorporating self-management capabilities. Self-management in AJS is based on the different types of optimization goals that may be encountered in application scheduling. For example, in a highly interactive system the goals of scheduling are related to response time and fairness; while batch systems are mainly concerned with throughput and turnaround time.

The autonomic features of AJS include self-configuration and self-optimization. Requests are stored in a container which reconfigures its overall capacity in a similar fashion to the autonomic container presented by Stevens et. al [12], (i.e., 80% full, increase capacity). Requests are serviced according to a high-level scheduling algorithm which can be changed at runtime according to the desired mode of the system – interactive or batch. A pool of service agents therefore work according to one of two implemented algorithms: (1) *first come first serve* (FCFS) - service jobs in the order they arrive, and (2) *shortest request next* (SRN) - service jobs in increasing order of required processing time. Self-optimization was incorporated by allowing the system dynamically add or remove service agents in a manner that is proportional to the workload of the system.

Recall that the purpose of AJS is to provide a more realistic context in which we can investigate self-testing in autonomic systems, and is not meant to be a solution or enhancement to the problem of job scheduling.

4.2 Architectural Design

Figure 2 shows the high-level architectural model for AJS, along with its integrated self-testing components. The autonomic job scheduling system (starting from the bottom-left of Figure 2) is comprised of: (1) managed resources for the job request container and service agent pool, represented by the subsystems `requestContainer` and `requestAgents` respectively; (2) copies of the aforementioned managed resources for testing purposes, represented by the subsystems `copyContainer` and `copyAgents`, (3) touchpoint autonomic managers for self-configuration and self-optimization, represented by the subsystems `rcSelfConfig` and `raSelfOptim` respectively; (4) an orchestrating autonomic manager, represented by the subsystem `oamScheduler`, and (5) a knowledge source, represented by the subsystem `ksScheduler`.

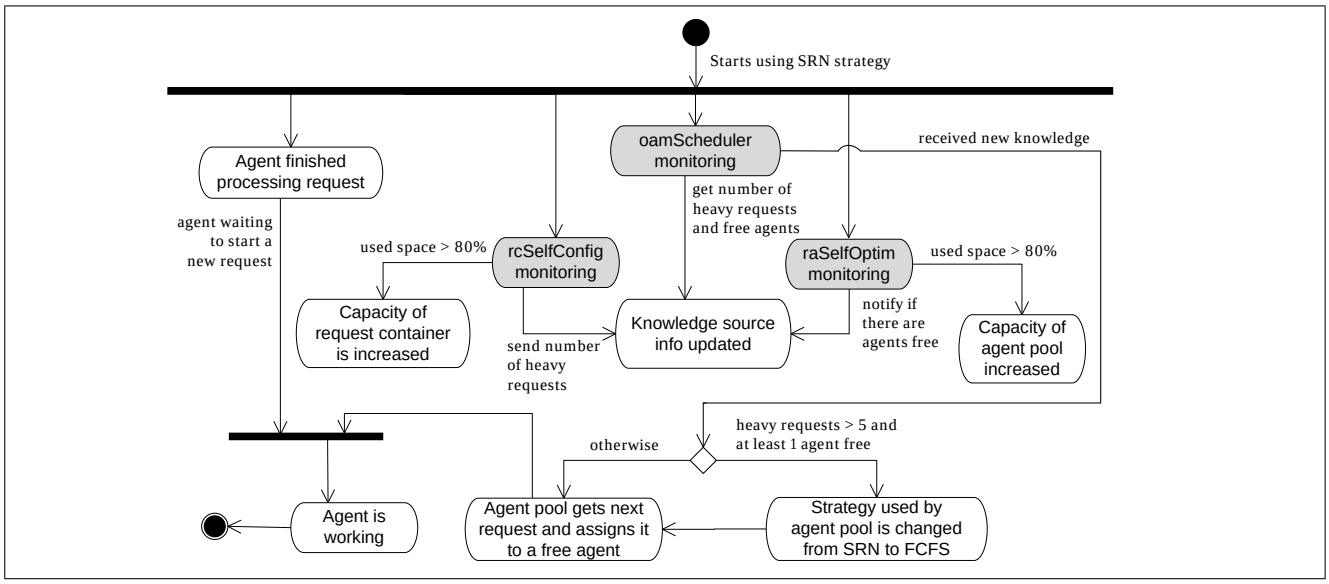

Figure 3: Activity diagram illustrating the self-management behavior of Autonomic Job Scheduler.

The autonomic system of AJS is monitored by its self-testing framework (shown on the right of Figure 2) to determine when change requests for the managed resources require validation. The self-testing framework consists of: (1) two touchpoint test managers `rcSelfTest` and `raSelfTest`, which perform testing on the resource copies as indicated by the dependencies on the managed resource layer of the autonomic system; and (2) an orchestrating test manager `otmScheduler`, which coordinates testing and interfaces with the autonomic system to initate the testing process as indicated by the dependency on the orchestrating autonomic manager subsystem.

5. PROTOTYPE

To validate the architectural design of AJS presented in Subsection 4.2, we developed a prototype in Java and used it as the focal point for our investigation of testing autonomic systems using replication with validation. The prototype was developed in Java 5.0 using the Eclipse 3.3 SDK, along with the necessary libraries for the tools to support testing and XML. In this section we provide the details of the prototype implementation including activity and object diagrams of various aspects of the system. We then discuss the lessons learned from developing the prototype, including the benefits gained from using the proposed architectural design and testing approach. The limitations of the prototype are also discussed in this section.

5.1 Activity Diagram

Figure 3 shows an activity diagram of the self-management behavior of the autonomic job scheduler prototype. Self-management in AJS includes: (1) increasing the capacity of the request container if it reaches 80 percent full capacity; (2) resizing the agent pool in proportion to the current workload of the system; and (3) swapping the scheduling algorithm between FCFS and SRN based on whether or not the ready queue contains heavy requests that will greatly reduce throughput. The aforementioned self-management symptoms of AJS are described in the context of the scenario of Figure 3 as follows:

- Assume the state of AJS begins running with the SRN algorithm enabled as shown at the top of Figure 3. In addition, we have some heavy requests in the ready queue and at least one available agent in the agent pool.
- The three autonomic management activities labeled with `oamScheduler`, `rcSelfConfig`, and `raSelfOptim` in Figure 3 (shaded region) are concurrently monitoring different aspects of the system.
- The touchpoint AM for the self-configuration of the request container `rcSelfConfig` continually increases the capacity of the request container every time the the used space exceeds 80 percent, while `raSelfOptim` does the same with the capacity of the agent pool. These two managers report to the knowledge source `ksScheduler` about the number of available agents and heavy requests.
- The orchestrating AM `oamScheduler` makes the decision as to whether or not the scheduling algorithm should be swapped based on the updated information in the knowledge source.
- The swapping algorithm symptom occurs when the number of heavy requests is greater than five and at least one agent is free.
- If the swapping algorithm symptom is not detected, the agent pool retrieves the next request and assigns it to a free agent.

5.2 Self-Management Subsystem

Figure 4 shows the minimal object diagram for AJS, including the primary objects from all communicating subsystems. All objects are labeled with the object name followed by a class type: `KnowledgeCoordinator` – an autonomic manager; `Touchpoint` – a manageability interface object; `ManagedResource` – a managed software resource, `Strategy` – a concrete scheduling strategy, or `Context` – a scheduling context.

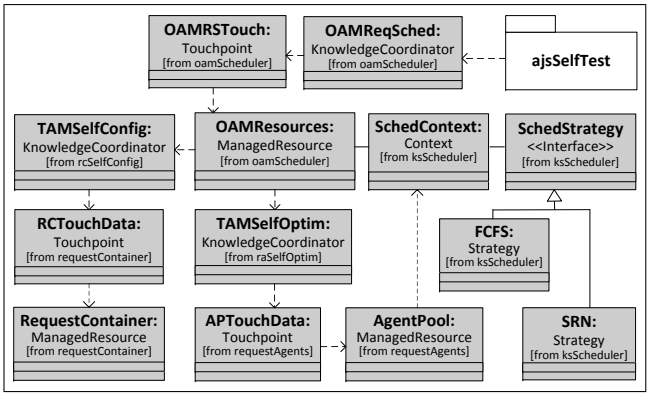

Figure 4: Minimal object diagram for AJS.

All autonomic managers for AJS were developed using the generic manager design presented in King et. al [9]. These managers are represented by the following objects in Figure 4: (1) `OAMReqSched` – Orchestrating AM, (2) `TAMSelfConfig` – Touchpoint AM for self-configuration, and (3) `TAMSelfOptim` – Touchpoint AM for self-optimization. Management policies were defined for each AM using the extensible markup language (XML), and an engine was developed to automatically parse the XML files and instantiate the managers with the appropriate policy information.

The manager `OAMReqSched` coordinates self-management activities using the object `OAMRSTouch`, which contains a reference to `OAMResources`. `OAMResources` encapsulates the two Touchpoint AMs as indicated by the dependencies on `TAMSelfConfig` and `TAMSelfOptim`, respectively. The Touchpoint AMs then use the touchpoint objects `RCTouchData` and `APTouchData` to manage the `RequestContainer` and the `AgentPool` resource objects, respectively. The scheduling algorithms were organized using the strategy design pattern [2]to allow dynamic reassignment of the active strategy, and facilitates the addition of new strategies in the future. The agent pool therefore invokes the object `SchedContext` that uses strategy interface `SchedStrategy` according to the context of system operation to call the appropriate concrete strategy objects `FCFS` or `SRN`. Due to space restrictions the objects of the self-test subsystem have been omitted from Figure 4 and represented by the package labeled `ajsSelfTest`. However, the details of the self-test component are described in the next subsection.

5.3 Self-Test Subsystem

The self-test subsystem implemented two test managers at the touchpoint level to dynamically validate changes to the managed resources, and one at the orchestrating level to coordinate testing. Similar to the self-management subsystem, the three manager objects `OTMSelfTest`, `SelfTestRC`, and `SelfTestAP` were developed using the generic manager design in [9]. Hence, the engine for automatically instantiating managers based on XML policies was reused for the test managers. In addition, Touchpoint TMs executed tests directly on the managed resources through the touchpoint objects `RCTouchData` and `APTouchData` in Figure 4.

Two tools were used to support the testing effort: JUnit, a Java unit testing tool from the xUnit family of testing frameworks and Cobertura, a Java code coverage analysis tool that calculates the percentage of source code exercised by unit tests. Test cases for the request container and agent pool were developed by extending the JUnit class `TestCase`, and stored in the classes `RCTest` and `APTest` respectively. The classes `RCCover` and `APCover` provide functionality for dynamically generating and executing batch files that set up Cobertura to instrument the request container and agent pool during testing.

5.4 Discussion

The main objective of this work was to provide a more complex prototype with which we could conduct our studies of testing autonomic systems using replication with validation. Applying our testing methodology in the context of a real problem provided greater insight into challenges of developing the autonomic system and the self-testing framework. Synchronization between components and ensuring harmony between the closed control loops were the major challenges experienced when developing the AJS application. Debugging the system was also quite difficult because of the heavy use of threads. However, we thought such a design was necessary to retain the independence of autonomic managers, test managers, and managed resources.

Our findings also revealed that the design the integrated self-testing framework [8] facilitates a high-level of reuse at the implementation stage. Test managers along with their validation policies can be viewed as special types of autonomic managers with the goal of testing the system. Therefore, a generic design for autonomic managers, test managers, and their associated policies greatly reduces the effort needed to develop both the autonomic software system and the self-testing framework. With the reusable monitor, analyze, plan, execute (MAPE) infrastructure already in place we could focus on defining the self-management and validation policies of the system. Limitations of the current prototype include the static lookup of predefined plans for both management and testing, which if incorporated would reflect the need for dynamic planning in self-testable autonomic software systems.

6. RELATED WORK

Several researchers have investigated areas related to the design and development of dependable self-managing systems [1, 12, 9, 3]. The work by Baldini et al. [1] is most closely related to our work; it describes a project that aims at the definition and implementation of an environment for dependable autonomic computing. They apply self-healing and self-testing techniques to increase the dependability of digital systems. Self-testing is achieved through the use of mobile agents, and the design and architecture of the approach is described. However, the description of the prototype does not provide low-level implementation details like those presented in this paper.

Stevens et al. [12] present the design and implementation details of a prototype of a self-testable autonomic system. The prototype introduces the notion of a self-testing autonomic container, which is a data structure that has the ability to configure and test during execution. They implement the underlying data structure as a stack, and simulate push and pop operations to add and remove random numbers from the stack. Our work differs from [12] in that we apply the approach of self-testing in the context of a realistic problem – job scheduling. We therefore provide the design and implementation details of a more complex application to investigate deeper problems related to the development

of self-testable autonomic systems.

King et al. [9] describe an integrated self-testing framework for autonomic computing systems. The work uses an extended prototype of the autonomic container developed by [12] to validate their testing methodology. The extended version of the autonomic container was implemented as a distributed system with the added feature of self-protection. Self-protection could be incorporated into our work to provide a more comprehensive application for investigating testing autonomic systems. However, although the remote stack prototype provides a more involved example than [12], it differs from our work in that it does not apply self-testing in the context of a real problem.

Huebscher et al. [3] provide the design of a simulation model of contexts as a means to test the context-logic of self-aware applications. This is achieved by allowing sensor data to be produced from a description of contexts, providing a highly flexible and configurable solution. Our work could benefit from the work by Huebscher et al. [3] with respect to monitoring environmental conditions for both self-management and self-testing symptoms.

7. CONCLUSION

In this paper we presented the object-oriented design and implementation of a self-testing autonomic job scheduler (AJS). The application featured a job scheduling simulation which was capable of self-configuration, self-optimization and self-testing. The self-testing component our application validated change requests using copies of managed resources. We used the prototype to further investigate the many challenges of testing autonomic systems at runtime.

Our future work involves incorporating adaptive features into AJS so that it can be used to study *safe adaptation with validation* – an approach to testing autonomic systems at runtime using the actual managed resources during the adaptation process.

ACKNOWLEDGMENTS are optional

8. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant IIS-0552555. The authors would like to thank their mentor Dr. Peter J. Clarke and the participants of the FIU REU Summer 2007 program for their contributions to this work.

9. REFERENCES

[1] A. Baldini, A. Benso, and P. Prinetto. A dependable autonomic computing environment for self-testing of complex heterogeneous systems. *Electronic Notes in Theoretical Computer Science*, 116(19):45–57, 2005.

[2] E. Gamma, J. Vlissides, R. Johnson, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[3] M. C. Huebscher and J. A. McCann. Simulation model for self-adaptive applications in pervasive computing. In *DEXA '04: Proceedings of the Database and Expert Systems Applications, 15th International Workshop on (DEXA'04)*, pages 694–698, Washington, DC, USA, 2004. IEEE Computer Society.

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

[5] IBM Corporation. IBM Research, 2002. http://www.research.ibm.com/autonomic/ (Sept. 2007).

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

[7] J. O. Kephart. Research challenges of autonomic computing. In *ICSE '05*, pages 15–22, 2005.

[8] T. M. King, D. Babich, J. Alava, R. Stevens, and P. J. Clarke. Towards self-testing in autonomic computing systems. In *ISADS '07*, pages 51–58, Washington, DC, USA, 2007. IEEE Computer Society.

[9] T. M. King, A. E. Ramirez, R. Cruz, and P. J. Clarke. An integrated self-testing framework for autonomic computing systems. *Journal of Computers*, 2(9):37–249, 2007.

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

[11] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[12] R. Stevens, B. Parsons, and T. M. King. A self-testing autonomic container. In *ACM-SE 45*, pages 1–6, New York, NY, USA, 2007. ACM Press.