

A Reusable Object-Oriented Design to Support Self-Testable Autonomic Software

Tariq M. King, Alain Ramirez, Peter J. Clarke
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
{tking003, aeste010, clarkep}@cis.fiu.edu

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

ABSTRACT

As the enabling technologies of autonomic computing continue to advance, it is imperative for researchers to exchange the details of their proposed techniques for designing, developing, and validating autonomic systems. Many of the software engineering issues related to building dependable autonomic systems can only be revealed by studying detailed designs and prototype implementations. In this paper we present a reusable object-oriented design for developing self-testable autonomic software. Our design aims to reduce the effort required to develop autonomic systems that are capable of runtime testing. Furthermore, we provide low-level implementation details of a case study, Autonomic Job Scheduler (AJS), developed using the proposed design.

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, OO Design, Self-Testing.

1. INTRODUCTION

Although autonomic computing (AC) has successfully stimulated the interest of many researchers and industry partners [4, 5, 6, 9], there is a general lack of detailed designs and freely available prototypes of projects based on autonomic computing. Such work products are highly valuable with respect to revealing many of the important software engineering issues that surround the construction of autonomic systems. To be autonomic, a system must incor-

porate features such as self-configuration, self-optimization, self-protection, and self-healing. Additional functionality is built into these types of systems to automate low-level tasks and allow administrators to specify management behavior as high-level policies [6]. The objective is to alleviate the burden of managing highly complex systems through the use of a computerized self-management infrastructure. However, this self-management infrastructure poses significant challenges with respect to design, development, and testing [7].

Dynamic changes and re-configurations resulting from self-management in autonomic systems are typically applied and accepted with little emphasis on validating them against the system requirements. Goals are set as high level policies that induce modifications to system components, and provide a way for system administrators to check if the system is operating within the bounds of some desired behavior. However, such an approach is insufficient to determine if the system behavior still conforms with the overall functional and non-functional requirements after such modifications have been implemented. Furthermore, the high level of automation in autonomic systems means that incorrect goal specification could yield potentially disastrous, and even irreversible effects on the components being managed. Runtime testing is therefore necessary in autonomic systems to validate their dynamic self-management features.

The object-oriented (OO) paradigm has become the de-facto standard for application development, thereby making it a good candidate for adoption by industry in the development of autonomic software. One of the major advantages of the OO approach is reusability [1, 3]. In this paper, we present a reusable OO design for developing self-testable autonomic software systems. Our design exploits the aforementioned types of OO reuse with the aim of reducing the effort required to develop autonomic systems that are capable of validating themselves.

The main contributions of this work are that it: (1) extends the dynamic test model for autonomic systems in [8] by applying the concepts of knowledge sources to testing activities, and explicitly describes the interdependency relationships of the model components; (2) provides a highly reusable detailed design for autonomic managers, test managers, touchpoints, and self-management policies that facilitates automation; and (3) elaborates on the implementation details of a case study developed using the proposed design approach, including challenges and lessons learned.

This paper is organized as follows: the next section contains background information on autonomic software. Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...\$5.00.

tion 3 presents the extended autonomic self-testing framework. Section 4 presents the proposed generic design for self-testable autonomic software systems. Section 5 provides details of the case study. Section 6 presents related work, and in Section 7 we conclude and discuss future work.

2. AUTONOMIC SOFTWARE

An autonomic computing (AC) system is a computing system characterized by one or more of the self-management features of autonomic computing, which include: self-configuration, self-optimization, self-protection, and self-healing. These systems manage themselves by automating low-level tasks while allowing administrators to specify the goals of the system as high-level policies. In this paper we restrict our discussion of an AC system to one in which all managed resources are software entities, disregarding any hardware components. We refer to such a system as an autonomic software system, or autonomic software. Autonomic software must be able to dynamically observe its own structure and behavior (*introspection*); adapt or modify its own structure and behavior (*intercession*) [13]; and be aware of their environmental and operational contexts.

Figure 1 shows the layered architecture of an autonomic software system which is composed of *managed resources*, *touchpoints*, *autonomic managers*, and *knowledge sources*. The bottom layer of managed resources in Figure 1 includes the software entities for which self-management services are being provided. Directly above the managed resources are manageability interfaces called touchpoints. Touchpoints implement sensor and effector behaviors [4, 6] that are used to automate low-level management tasks. Sensors provide introspection mechanisms for gathering details on the current structure or behavior of managed resources, while effectors provide intercession mechanisms to facilitate structural or behavioral modifications.

A higher level of management is provided by autonomic managers (AMs). There are two categories of AMs, namely Touchpoint AMs and Orchestrating AMs [6]. Touchpoint AMs work directly with managed resources through their touchpoint interfaces. Orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for individual resources. The topmost layer is an implementation of a management console, called the manual manager, which facilitates the human administrator activity. The vertical layer of knowledge sources implements registries or repositories that can be used to extend the capabilities of AMs, and may be directly accessed via the manual manager.

Self-management in autonomic software is realized through a series of intelligent closed control loops [4] within autonomic managers. Figure 2 shows the structure of autonomic managers with respect to these closed control loops, which are generally implemented as *monitor*, *analyze*, *plan*, and *execute* (MAPE) functions. The monitor function collects state information from the managed resource and correlates them into symptoms for analysis. If analysis determines that a change is needed, a change request is generated and a change plan is formulated for execution on the managed resource. AMs also contain a knowledge component which allows access to data shared by the MAPE functions. This built-in knowledge component generally contains information relating to self-management policies, and interacts with the knowledge sources in Figure 1 to provide an extensible self-management framework.

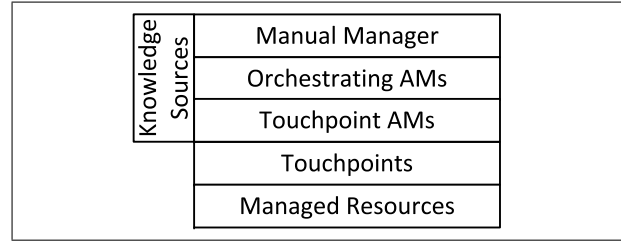


Figure 1: Architecture for autonomic software [4].

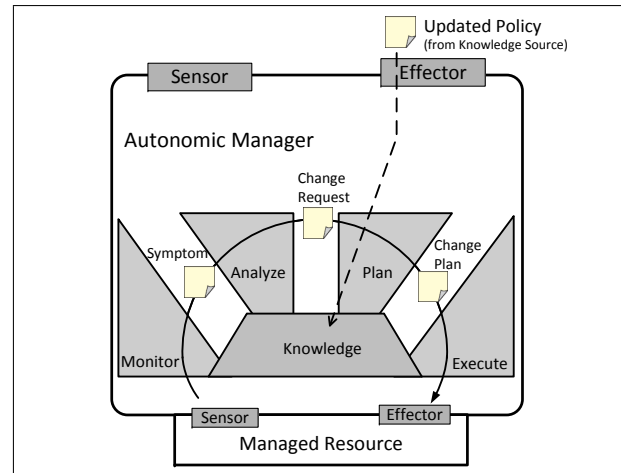


Figure 2: Structure of autonomic managers [4].

3. AUTONOMIC SELF-TESTING

In this section we provide an overview of the dynamic test model for autonomic systems proposed by King et al. [8], and extend it to include knowledge sources for testing activities in autonomic software. We describe the components of the extended test model, and outline their responsibilities and inter-dependency relationships. We also discuss the commonalities between an autonomic software system and the proposed test model for autonomic systems.

King et al. [8] presented a self-testing framework for autonomic computing systems based on two strategies – *safe adaptation with validation* and *replication with validation*. Note that the workflow of these two approaches is already described in [8] and therefore is not within the scope of this paper. Instead, we focus on extending the self-testing framework and identifying the dependency relationships imposed by the workflow.

Figure 3 shows the extended architecture for a self-testing autonomic software system, based on the dynamic high-level test model in [8]. The autonomic software system is shown to the left of Figure 3 and the self-testing framework is on the right, while the dotted arrows represent dependency relationships between communicating components. Recall from Section 2 that we restrict our discussion of an autonomic systems to exclude hardware, and therefore all of the components of Figure 3 are software entities.

The self-testing framework is composed of test managers (TMs), and test knowledge sources that collaborate to provide validation services to the autonomic software system. Like autonomic managers, TMs may also be Orchestrating

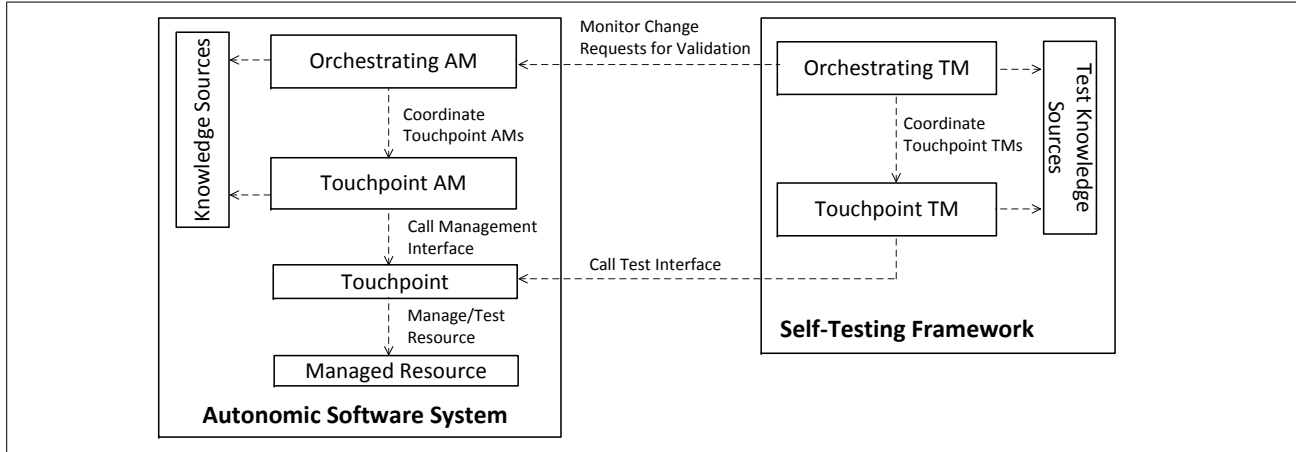


Figure 3: Architecture of self-testing autonomous software system based on the dynamic test model in [8].

or Touchpoint. Orchestrating TMs interface with the autonomic software system, and coordinate the high-level testing activity by managing Touchpoint TMs; as indicated by the dependencies on the Orchestrating AM and Touchpoint TM layers of Figure 3, respectively. Touchpoint TMs perform low-level testing tasks directly on managed resources using the touchpoint layer of the autonomic system. These low-level testing capabilities include the ability to control the internal state of the managed resource and observe its output, which may already be provided through the management interface. To perform their testing duties, TMs are composed of components that implement intelligent control loops that are consistent with the MAPE structure of AMs [8].

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. 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 updated test information.

The extended self-testing framework applies the concepts of autonomic managers, and knowledge sources to testing activities in autonomic software. TMs are structurally equivalent to autonomic managers with respect to hierarchical organization and internal composition. In addition, test knowledge sources require mechanisms similar to those in knowledge sources with respect to storing, accessing, and registering information about autonomic behaviors. In essence, the only difference between the self-testing framework and the autonomic software system is in the actual goal-specific details of the information that needs to be stored within the respective components. Therefore, time and resources can be saved when developing self-testable autonomic software by harnessing a generic reusable design to implement both the autonomic software system and the self-testing framework.

4. GENERIC DESIGN

Stevens et. al [11] presented a prototype of a self-testing autonomic container to introduce the notion of self-testing in autonomic computing systems. The prototype used a stack of random numbers that automatically configures itself at runtime by increasing its capacity, and then dynamically validates the re-configuration. In this section we use the idea of a self-configuring container as an illustrative example to demonstrate various aspects of the proposed design.

4.1 Touchpoints

As previously mentioned, the lowest level of self-management in autonomic software systems involves autonomic managers which directly interact with managed resources through interfaces called touchpoints [4]. We propose to design touchpoints as objects that encapsulate the managed resource by holding its object reference. Sensor and effector methods can then be implemented within the touchpoint class using the object reference of the managed resource to dynamically collect the relevant statistics, and invoke the appropriate set of self-management actions.

Designing touchpoints as classes provides an extensible layer of abstraction above the managed resource level, which allows sensors and effectors to be tailored to the specific management requirements. For example, the public interface of the underlying stack resource of autonomic container [11] provides the methods `getSize()` and `getCapacity()` for retrieving the number of elements currently in the stack and its overall capacity, respectively. However, the self-management infrastructure of the container is only concerned with checking the ratio of the stack size to its overall capacity. Therefore, the touchpoint class for an autonomic container with a stack reference `s` could implement a method `getFullPercent()` that returns the value of the expression: $(s.getSize() / s.getCapacity()) * 100$. This allows autonomic management symptoms to be expressed in a more concise and natural manner, e.g., `getFullPercent() >= 80`.

4.2 Autonomic Managers

Figure 4 depicts the proposed design of autonomic managers and test managers in a self-testable autonomic software system. The package `edu.fiu.strg.ACSTF.manager`

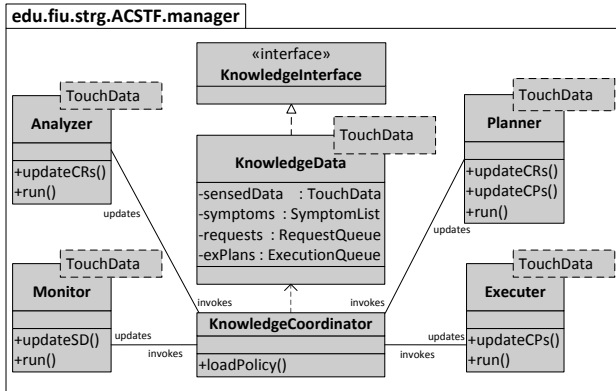


Figure 4: Reusable autonomic manager design.

shows the class design, which uses synchronized threads, generics, and reflection in Java [12] to implement the MAPE functions and shared knowledge. MAPE functions are encapsulated in the **Monitor**, **Analyzer**, **Planner**, and **Executer** classes shown in Figure 4, which will operate as independent threads. Each MAPE class is parameterized with the template class **TouchData**, which represents the touchpoint class that will be used to carry out self-management.

A controller class, **KnowledgeCoordinator** in Figure 4, is responsible for coordinating the MAPE objects with respect to initialization and synchronization. When a manager is instantiated, the **KnowledgeCoordinator** loads a policy file containing the following information: (1) the fully qualified class name of the touchpoint object to be used; (2) the method name of the sensor function to be polled by the monitor; (3) a set of symptoms defined by relational mappings between the variables of the touchpoint and desired values; and (4) a set of executable change plans represented by sequences of effector methods to be executed and their actual parameters.

The knowledge repository of the manager, represented by the class **KnowledgeData** in Figure 4, realizes the interface **KnowledgeInterface** that is used by the MAPE functions to access shared information. Queues to hold newly generated change requests and change plans to be executed have also been incorporated into the knowledge component. Producer-consumer relationships are used within the manager so that as soon as a request or plan is generated by one function, it is handled by its neighboring function in the intelligent control loop.

4.3 Management Policies

The design for high-level policies uses the Extensible Markup Language (XML) format, along with a standard encoding scheme. Each policy is designed with name and version attributes to distinguish it from other policies, and facilitate automation with respect to upgrades. A management policy for the self-configuration feature of our illustrative example is shown in Figure 5. Policies are divided into three major sections represented by `<monitor>`, `<analyzer>`, and `<planner>` tags. Each section contains the portion of knowledge that will be primarily used by the component corresponding to its tag name. For example, combining the values of the attributes `touchpackage`, `touchclass`, and `sensormethod` of the `<monitor>` tag in Figure 5, produces the fully qualified method name of the sensor function used

```
<policy name="TAM Self-Config" version="1.0">

  <monitor touchpackage="edu.fiu.strg.STAC"
    touchclass = "TouchStack"
    sensormethod="getState" />

  <analyzer>
    <symptom sid="TAM_SC_001">
      <mapping var="getFullPercent"
        op ">="
        val="80" />
    </symptom>
  </analyzer>

  <planner>
    <plan pid="TAM_SC_001">
      <action effector="increaseCapacity" />
    </plan>
  </planner>
</policy>
```

Figure 5: Design of management policies.

by the monitor of the Touchpoint AM for self-configuration. This method collects dynamic state information on the underlying stack data structure of the autonomic container. Polling would be achieved by first passing the fully qualified class name `edu.fiu.strg.STAC.TouchStack` to the monitor as the template parameter `TouchData`, shown in Figure 4. The monitor then uses reflection to instantiate the class and continuously invokes the `getState` method.

The state information returned from the invocation of the sensor method is compared with one or more predefined symptoms, which were loaded from the `<analyzer>` tag of the policy. Figure 5 shows a symptom with its identifier (`sid`) attribute set to `"TAM_SC_001"`. This symptom consists of a single mapping between the touchpoint state variable `"getFullPercent"` and the value `"80"`, using the relational comparison operator `">="`. The `getFullPercent` variable is defined in terms of the state of the stack object as described in Section 4.1. Our policy design is highly extensible as it allows multiple variable mappings to be defined for a single symptom, and multiple symptoms to be defined for any manager.

Symptoms are associated with plans which provide remedies as sequences of low-level management tasks. Plans consist of a plan identifier (`pid`) attribute that is logically linked to a symptom identifier, along with one or more actions. Figure 5 shows a plan with `pid "TAM_SC_001"`, which corresponds to the previously mentioned symptom for detecting self-configuration requests. The actions of the plan consist of the single effector method `increaseCapacity`, which increases the size of the stack in anticipation of depleting the available storage slots. Actions may also contain one or more parameter tags (not shown in Figure 5) that provide the actual parameters to be used when executing the effector methods. In a similar fashion to the monitor, the executer component uses Java reflection to invoke the effector methods with the specified parameters.

5. CASE STUDY

To show the feasibility of the proposed approach, we applied our design to the development of a self-testable autonomic software system. Our case study applies the concept

of an autonomic container in a more realistic context, which we refer to as an *Autonomic Job Scheduler* (AJS). Recall that the autonomic container developed by Stevens et al. [11] implemented self-configuration into a stack of random numbers. We have extended the concept of an autonomic container to one which holds a collection of job requests, and a pool of simulated software agents that can independently remove and handle those requests. In this section we describe the main features of Autonomic Job Scheduler, and provide details on its implementation. We then discuss the lessons learned from developing the case study, including the advantages of using a generic design to develop the autonomic features and self-testing framework. The major challenges and limitations of the study are also discussed in this section.

5.1 Main Features

AJS simulates activities such as job submission, servicing and monitoring of job executions, while incorporating self-management and self-testing capabilities. 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 [11], (i.e., 80% full, increase capacity). Servicing of requests is performed according to a high-level scheduling algorithm which can be changed at runtime. Therefore, service agents in the agent pool work according to one of the 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. The ability to dynamically swap scheduling algorithms provided an additional self-configuration feature for our application. Self-optimization was incorporated by dynamically adding/removing agents to/from the agent pool in a manner that is proportional to the workload of the system.

5.2 Self-Management Subsystem

AJS 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. Figure 6 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** – generic manager; **Touchpoint** – manageability object type; **ManagedResource** – managed software entity, **Strategy** – concrete strategy, or **Context** – scheduling context.

All autonomic managers for AJS were developed using the generic manager design presented in Subsection 4.2. These managers are represented by the following objects in Figure 6: (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 proposed XML design, 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**

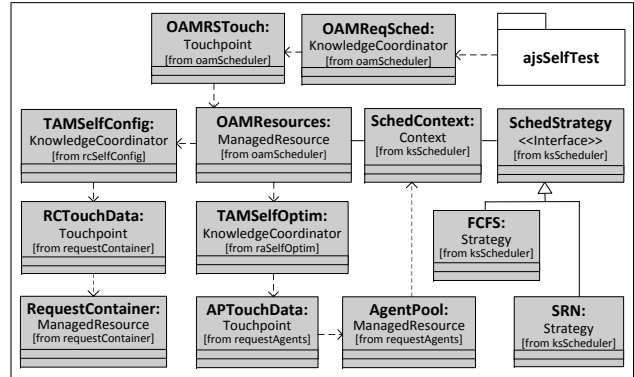


Figure 6: Minimal object diagram for AJS.

and **APTouchData** to manage the **RequestContainer** and the **AgentPool** resource objects, respectively. The scheduling algorithms were organized using the strategy design pattern [3] 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 6 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 **OTMSTest**, **SelfTestRC**, and **SelfTestAP** were developed using the proposed generic manager design and policy design. 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 6.

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 conducting the case study was to provide justifications for the use of a generic design when developing self-testable autonomic software. When compared to previous development experiences of autonomic software prototypes, our design greatly reduced the effort needed to develop both the autonomic software system and the self-testing framework. Encapsulating the touchpoints

as objects made it easy to set multiple managers to administrate and test different resources by simply passing around the touchpoint reference. The manager design was especially useful as having a generic MAPE infrastructure already in place meant that we could focus on defining the self-management and validation policies of the system. Furthermore, the engine for parsing the policy files and automatically instantiating the managers was highly reusable.

The case study also validates the architectural model for developing self-testable software systems shown in Figure 3 in a more realistic context. 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. 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

Although the research community would clearly benefit from detailed designs and implementation of autonomic computing projects, few researchers have addressed this issue when describing their projects and prototypes.

Several researchers have investigated areas related to the design and development of dependable self-managing systems [2, 8, 10, 11]. The work by Serugendo et al. [2] is most closely related to our work; it describes a generic framework which supports the development of trustworthy self-adaptive and self-organising systems. The framework facilitates decision-making at design-time and at runtime. Runtime decision-making strongly supports the need for dynamic planning mechanisms in self-testable autonomic software. Such techniques can easily be incorporated into our design to improve the overall approach. Serugendo et al. [10] also describes initial proof-of-concept prototypes that provide enabling technologies to realise their proposed framework. However, the description of the prototypes does not provide low-level design and implementation details like those presented in this paper.

Patouni et al. [10] present a framework for the design and deployment of autonomic components. They introduce the definition of a generic autonomic component, and describe the dynamic composition and replacement of these component. The case study presented in our paper does not incorporate mechanisms for dynamic adaptation of components. Developing such a prototype can provide valuable information with respect to the design and implementation of adaptive autonomic software, and be used to further validate our testing methodology.

Stevens et al. [11] 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 simu-

late push and pop operations to add and remove random numbers from the stack. Our approach differs from [11] in that we apply a reusable design to the construction of the autonomic container. Furthermore, we provide the design and implementation details of a more complex application to investigate deeper problems related to the development of self-testable autonomic software.

7. CONCLUSION

In this paper we proposed a reusable OO detailed design for building self-testable autonomic software systems. Our design incorporates several language support features such as generics, reflection, and multi-threading to aid the development of the autonomic system and its self-testing framework. To show the feasibility of our approach, we developed a case study that applies the features of self-configuration, self-optimization, and self-testing in the context of job scheduling. Our future work involves extending the case study to incorporate dynamic adaptation.

8. REFERENCES

- [1] S. Ambler. A realistic look at object-oriented reuse. *Dr. Dobbs's Journal*, January 1998.
- [2] G. Di Marzo Serugendo, J. Fitzgerald, A. Romanovsky, N. Guelfi. A generic framework for the engineering of self-adaptive and self-organising systems. Technical report, University of Newcastle, Newcastle, UK, April 2007.
- [3] 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.
- [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] H. A. Muller, L. O'Brien, M. Klein, and B. Wood. Autonomic computing. Technical report, Carnegie Mellon University and SEI, April 2006.
- [10] E. Patouni and N. Alonistioti. A framework for the deployment of self-managing and self-configuring components in autonomic environments. In *WOWMOM '06*, pages 480–484, Washington, DC, USA, 2006. IEEE Computer Society.
- [11] 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.
- [12] Sun Microsystems, Inc. Core Java J2SE, February 2005. <http://java.sun.com/j2se/> (Sept. 2007).
- [13] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley. Enabling safe dynamic component-based software adaptation. In *WADS*, pages 194–211, 2004.