Design Patterns for Developing Dynamically Adaptive Systems *

Andres J. Ramirez and Betty H.C. Cheng
Michigan State University
3115 Engineering Building
East Lansing, Michigan 48824
{ramir105, chengb}@cse.msu.edu

ABSTRACT

As applications grow in size and complexity, and computing infrastructure continues to evolve, it becomes increasingly difficult to build a system that satisfies all requirements and constraints that might arise during its lifetime. As a result, there is an increasing need for software to adapt in response to new requirements and environmental conditions after it has been deployed. Due to their high complexity, adaptive programs are generally difficult to specify, design, verify, and validate. In addition, the current lack of reusable design expertise that can be leveraged from one adaptive system to another further exacerbates the problem. To address this problem, we studied over thirty adaptation-related research and project implementations available from the literature and open sources to harvest adaptation-oriented design patterns that support the development of adaptive systems. These patterns facilitate the separate development of the functional logic and the adaptive logic. We present these design patterns within the context of a modeling-based development process for dynamically adaptive systems. In order to address the assurance of these adaptive systems, the patterns also include templates for formally specifying invariant properties of adaptive systems.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Design Patterns; D.2.10 [**Design**]: Methodologies

Keywords

Design Patterns, Adaptive Systems, Autonomic Systems

*This work has been supported in part by NSF grants CCF-0541131, CNS-0551622, CCF-0750787, IIP-0700329, and CCF-0820220, Army Research Office W911NF-08-1-0495, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Ford Motor Company, and a grant from Michigan State University's Quality Fund.

ICAC '09 Barcelona, Spain

1. INTRODUCTION

As applications grow in size, complexity, and heterogeneity in response to growing computational needs, it is increasingly difficult to build a system that satisfies all requirements and design constraints that it will encounter during its lifetime. Many of these systems are required to run continuously, disallowing long downtimes where humans look for places to modify the code. As a result, it is important to be able to adapt an application's behavior at run time in response to changing requirements and environmental conditions [21]. IBM proposed autonomic computing [16] to meet this need, where a system manages itself based on high-level objectives from a systems administrator, thus promoting self-management and self-reconfiguration. Due to their high complexity, adaptive and autonomic systems are generally difficult to specify, design, verify, and validate [32]. In addition, the current lack of reusable design expertise that can be leveraged from one adaptive system to another further exacerbates the problem. To address this problem, we studied over thirty adaptation-related research and project implementations available from the literature and open sources to harvest and develop adaptation-oriented design patterns that support the development of adaptive systems. This paper describes the harvested adaptation design patterns and how they can be used to construct adaptive and autonomic systems.

Most adaptive systems, including autonomic systems, comprise three key elements: monitoring, decision-making, and reconfiguration. *Monitoring* enables an application to be aware of its environment and detect conditions warranting reconfiguration; *decision-making* determines what set of monitored conditions should trigger a specific reconfiguration response; and *reconfiguration* enables an application to change itself in order to fulfill its requirements. Not only must developers design and implement each of these elements correctly, they must also carefully determine their interactions. For instance, if the monitoring process fails to report a significant environmental change, then the decision-making process may incorrectly trigger an unnecessary (or even detrimental) reconfiguration. Unfortunately, until recently, most approaches have addressed adaptation using *ad hoc* techniques [10]. To address these concerns, researchers have developed adaptation-enabling frameworks [2, 9], middleware [18, 23], and language-based support [28]. These approaches, however, tend to be tightly coupled with specific domains or technologies, thus limiting their applicability with respect to the problem being addressed. Design patterns, on the other hand, work at the modeling and de-

sign level of abstraction, thereby potentially increasing the amount of design reuse when compared to other approaches.

This paper presents twelve adaptation-oriented design patterns to facilitate the reuse of adaptation expertise. In the spirit of the original design patterns by Gamma *et al.* [8], each of the adaptation-oriented design patterns were developed by generalizing several existing design solutions. For each design pattern, we use platform-independent models to represent the solution. In addition, by focusing on the recurring challenges found in monitoring, decision-making, and reconfiguration activities, our design patterns separate the adaptive logic from the functional logic. This separation of concerns facilitates the reuse of adaptation designs across multiple applications and domains. Similarly, while harvesting each candidate design, we have observed recurring interactions between monitoring, decision-making, and reconfiguration processes. This information enables us to suggest which design patterns should be used together. Lastly, we extended the design pattern template introduced by Gamma *et al.* [8] with a *Behavior* and *Constraints* fields. Constraints contain specific templates that can be used to specify invariant properties that must be satisfied by state-based models from the Behavior field once the design pattern is instantiated. Since our approach is compatible with the high assurance model-based development process for adaptive systems previously introduced by Zhang and Cheng [32], automated verification techniques can be used to analyze the instantiated design patterns against functional and adaptation-specific properties.

Harvesting design patterns is a difficult and subjective task for two main reasons. First, it is impractical to examine all available systems and research projects associated with adaptation. Second, some of the surveyed systems had little to no documentation accompanying their design. To ensure that the design patterns harvested were sufficiently mature to aid developers in building adaptive systems, we performed two forms of validation in this work. First, we found additional instances of our design patterns in other adaptive systems. In some cases, information from these new instances enabled us to further generalize the solutions and refine the design patterns. Second, we re-engineered an adaptive news web server, originally developed with an object-oriented framework for adaptive systems by Garlan *et al.* [3], from scratch using our design patterns. This case study was used to compare and contrast a framework versus design pattern-based approach for developing adaptive systems.

From this work, we see that recurring design solutions are being independently used by the adaptive systems community. Our pattern collection provides a resource for developers to take advantage of this experience. Additionally, instantiated versions of our design patterns are amenable to automated analysis to ensure a design satisfies certain properties. The remainder of this paper is organized as follows. Section 2 presents background information including an overview of design patterns and an introduction to the Zhang-Cheng model-based development process [32]. Section 3 presents a survey of related work. Section 4 explains how we created these patterns, provides a table listing all patterns found thus far, the adaptation pattern template and an example pattern. Section 5 presents a proof of concept case study that applies several of our patterns in order to re-engineer an adaptive web server. Section 6 compares our pattern-based approach with a framework-based approach. Lastly, Section 7 presents our main findings and discusses future directions of work.

2. BACKGROUND

This section briefly introduces two key topics fundamental to this paper, design patterns and the model-based development process by Zhang and Cheng [32].

2.1 Design Patterns

A design pattern is a general and reusable solution to a commonly recurring problem in design [8]. A software design pattern does not provide code, nor can it be directly transformed into code. Instead, a design pattern identifies and abstractly models the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. Each design pattern has four essential elements [8]. A *pattern name* is a handle that can be used to describe a design pattern, its solutions, and consequences. The *problem* describes under what context to apply the design pattern. The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. Lastly, the *consequences* describe the results and tradeoffs of applying the design pattern. Each of these four essential elements are organized and presented in the design pattern template *fields*.

2.2 Model-based Development Process

Zhang and Cheng [32] previously introduced a model-based development process with the objective of guiding the rigorous development of adaptive programs. The process separates the adaptive behavior and the non-adaptive behavior specifications of adaptive programs. By doing so, the respective models are easier to specify and more amenable to automated analysis, visual inspection, and modification. As Figure 1 illustrates, the process starts with high-level system goals (G) and progresses through design models (M_i, M_j) to code. The focus of the process is the specification of key properties (e.g., Φ_i, Φ_j) at each of the major development phases. While the original work used Petri-nets to illustrate the design phase, the process itself is compatible with other state-based modeling approaches such as the Unified Modeling Language (UML) [27].

The model-based development process comprises six key steps (Figure 1):

1. Specify global properties, INV, using a high-level specification language such as temporal logic.

2. Identify the different domains, D_i, or environmental conditions under which a program with requirements R_i will execute.

3. Using a high-level specification language, specify local properties, Φ_i for each domain identified in step (2).

4. Build state-based models (M_i and M_j) of the non-adaptive programs in each domain. Simulations and verifications can be applied to verify and validate the models against both the local (Φ_i, Φ_j) and global properties (INV) previously specified.

5. Identify the possible scenarios in which dynamic changes may occur. Build adaptive models, $M_{i,j}$ and $M_{j,i}$, to

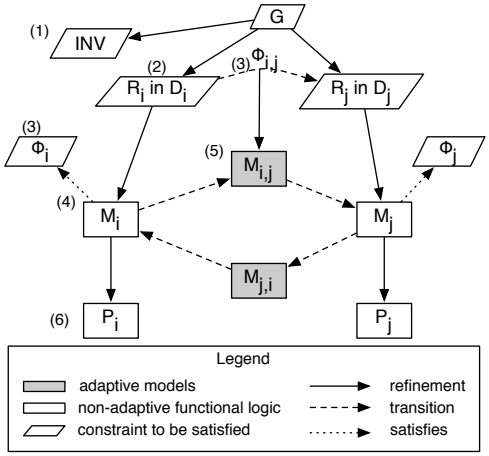

Figure 1: Model-based Development Process.

safely transfer execution from a source program to a target program. Specify *transitional properties*, $\Phi_{i,j}$ and $\Phi_{j,i}$, to indicate the properties that must be satisfied *during* the adaptation process. As with step (4), simulations and verifications can be applied to verify and validate the adaptive models against global and transitional properties [33].

6. The state-based models can be used to either generate rapid prototypes or to guide the development of adaptive programs [32].

3. RELATED WORK

This section overviews selected efforts conducted by researchers to facilitate the development of dynamically adaptive software. Although not exhaustive, the techniques and projects included in this section are the most relevant with respect to our adaptation-oriented design patterns; some of these projects served as a resource for our harvesting process.

Middleware. Recently, researchers have focused on extending middleware approaches to provide adaptation services [4, 11, 23]. Middleware refers to the various layers of services that separate applications from operating systems and network protocols [21]. The different service layers of adaptive middleware serve as a level of indirection by intercepting and modifying messages as needed. Two examples of middleware approaches for adaptation include the Mobility and ADaptability enAbling Middleware (MADAM) [23] and the Adaptive CORBA Template (ACT) [28]. The MADAM project provides a general component model and middleware infrastructure that supports various adaptation styles for *mobile* applications. ACT enables run time improvements to CORBA applications in response to changing requirements and environmental conditions [28] by weaving adaptive code into an object request broker (ORB) at run time. One benefit of middleware-based adaptation approaches, such as MADAM and ACT, is that they shield developers from dealing with resource distribution, component probing, and application reconfiguration, thus alleviating complex tasks previously relegated to developers. However, middleware tends to be highly domain-specific and, as a result, may not be readily applicable for many systems.

Frameworks. Adaptive software research has developed frameworks for building adaptive systems [2, 9]. A framework is a set of cooperating classes that make up a reusable design for a specific class of software [8]. Among other objectives, the framework dictates the overall architecture of the application and its thread of control, thus leading to an inversion of control where developers write code that gets called by the framework. Rainbow is an architecture-based self-adaptation framework with reusable infrastructure [3, 9]. Rainbow supports distributed component monitoring, probe and gauge deployment, architectural-based system representation and adaptation strategies, and effectors to reconfigure the system. A benefit of adaptation-oriented frameworks is that it provides large amounts of reusable code, thereby enabling developers to build adaptive applications more rapidly. Nevertheless, some creative freedom is lost because many design decisions have already been made by the framework developers [8]. Additionally, framework-based applications are sensitive to changes in the framework's interface.

Design Patterns. Gomaa *et al.* proposed several patterns for dynamically reconfiguring specific types of software architectures at run time [10]. In particular, they extended the concepts of dynamic change management introduced by Kramer and Magee [19] by introducing four design patterns to specify the behavior required to dynamically reconfigure master/slave, centralized, server/client, and decentralized architectures. Gomaa's patterns identify when it is safe to perform a reconfiguration based on inherent characteristics of an application's architecture. One of the key contributions of this work is the identification of when it is safe to perform a reconfiguration based on the inherent characteristics of an application's architecture. Gomaa *et al.* uses hierarchical UML state diagram templates to depict the necessary behavior for reconfiguring system architectures in high-level terms. Although these patterns are helpful to developers implementing dynamically adaptive systems, their contents are not organized in Gamma's template format and the safeness of when to apply an adaptation is the focus of their assurance concerns.

4. ADAPTATION DESIGN PATTERNS

This section highlights the main results of our design pattern harvesting efforts. First, we provide a brief overview of the harvesting process used for developing each of our adaptation design patterns. We then present a table of the adaptation design patterns harvested thus far, including their names, classifications, descriptions of their purposes, and a list of selected sources. Lastly, we present our adaptation design pattern template and one of our adaptation design patterns is used to illustrate the intent of each field.

4.1 Harvesting Process

Harvesting design patterns is a difficult and subjective process for two main reasons. First, there is no standard methodology for harvesting and developing design patterns in practice. Second, there is no existing set of metrics that quantify the quality of a resulting design pattern. As a result, we do not *advocate* a particular process for harvesting design patterns, instead we simply *document* the process we used to develop our adaptation design patterns.

Identify Problem. The first step in the harvesting process involves identifying and defining a recurring problem

related to adaptation. One way to identify recurring problems is to analyze research publications with common topics related to monitoring, decision-making, and reconfiguration. Once a recurrent problem has been identified, the intent, context, and motivation for addressing the problem as a design pattern must also be determined. A clear definition of these fields will help narrow the search for existing solutions to the recurring problem. Next, developers need to select the relevant data sources that will be analyzed and generalized into design patterns. Three types of data sources are available for this task: commercial applications, open-source implementations, and research projects. In general, we restricted our sources to include only open-source and research project implementations given their ease of accessibility.

Generalize Solution. Once a problem has been defined and various sources of data have been selected, the artifacts must then be abstracted and generalized into one representative design pattern solution. Several steps must be taken to abstract and unify various solutions into a single design pattern. First, structural and behavioral similarities and differences must be identified between the various solutions. These include discovering objects, their interactions, associations, responsibilities, multiplicities, constraints, etc. Once various solutions have been unified into model diagrams that captures the generic solution to the recurring problem, then this model can be further generalized and put into template form as more instances are found.

Validate Design. We used two forms of validation to assess the resulting design patterns. First, we found additional instances of our design patterns in other adaptive systems. Each new instance further strengthened the validity of the solution as well as provided additional information for refining and generalizing the design pattern, especially during the early stages of the harvesting process. Second, we applied a subset of the harvested design patterns to a case study application. Instantiating the harvested design patterns enabled us to assess whether the level of detail provided in each pattern was sufficient to guide the development of an adaptive system.

4.2 Catalogue of Design Patterns

Table 1 gives an enumeration of the twelve adaptation design patterns harvested thus far, their classifications, a brief descriptions of their purpose, and a list of selected sources used for harvesting. We have limited the scope of our design patterns to only address the software side of adaptive systems because software is more amenable to dynamic adaptation than hardware. Each of our design patterns can be classified as monitoring (M), decision-making (DM), or reconfiguration (R) based on their overall objective. In addition, monitoring and decision-making design patterns can also be classified as either creational (C) or structural (S), as defined by Gamma *et al.* [8]. Likewise, reconfiguration design patterns can also be classified as behavioral (B) and structural since they specify how to physically restructure an architecture once the system has reached a *quiescent* or safe state for adaptation. Although Gomaa *et al.* previously introduced four software reconfiguration patterns [10], their goal was to identify states from which various architectures could be safely reconfigured. While we leveraged two of those patterns, we significantly extended them. Specifically, we provided information for them for all the pattern fields, catalogued them in our pattern template, provided several constraints that address safety and liveness properties in both LTL and A-LTL templates, and indicated which related patterns can be used together. We also introduced two new reconfiguration patterns that can be used for structural-based reconfiguration beyond the architectural styles presented by the Gomaa patterns [10].

It is important to consider the following issues when evaluating the set of adaptation design patterns. First, these patterns are not intended to be exhaustive. We anticipate that a number of other domain-specific design patterns are likely to be harvested as the adaptation technology continues to mature; for example, the mobile computing domain has made extensive use of adaptive technology, and, as a result, it is likely that mobile computing domain-specific design patterns can also be harvested. Second, the set of adaptation design patterns presented in this paper only includes designs that have been applied successfully to at least two different adaptive systems. Third, although most of our adaptation design patterns can be applied to a wide range of adaptive systems, some are applicable only to domain-specific adaptive systems. The *Intent* section should be consulted to determine the applicability of each design pattern given a particular application and domain. In addition, the *Related Patterns* section provides information regarding which design patterns are commonly used in conjunction. Complete details for each design pattern, including a more comprehensive list of sources, can be found in [25].

4.3 Adaptation Design Pattern Template

This paper uses a template similar in style to that used by Gamma *et al.* [8] in order to facilitate the understanding and application of the adaptation design patterns. We have modified a few of the original design pattern template fields to address the specific needs of adaptive systems. First, the *Known As*, *Implementation*, and *Sample Code* fields have been removed. The *Known As* section is not applicable as, to the best of our knowledge, the majority of the design patterns presented in this paper, with the exception of the Gomaa patterns [10], have not been previously documented. Likewise, the *Implementation* and *Sample Code* sections are too specific for the design patterns presented in this paper. Second, the template has been extended with *Behavior* and *Constraints* sections. The *Behavior* section presents either sequence and/or state diagrams that illustrate sample behavior. The *Constraints* section uses Linear Temporal Logic (LTL) and Adapt-Operator LTL (A-LTL) [31], annotated with textual descriptions to specify properties that must be satisfied by the instantiated design patterns. Lastly, while Gamma *et al.* used the Object Modeling Technique (OMT) to represent structural and behavioral diagrams, we used the Unified Modeling Language (UML) to give structural and behavioral information about each design pattern. Specifically, structural diagrams are represented through UML class diagrams (for monitoring and decision-making patterns) and UML component diagrams (for reconfiguration patterns). Likewise, UML statecharts are used to depict a pattern's behavior.

We now present our adaptation-oriented design pattern template and the Sensor Factory pattern as an example to illustrate the information presented in the various fields. The sans-serif font is used to denote the information from the Sensor Factory pattern. Note that some of the descriptions have been elided due to space constraints.

Table 1: Catalogue of Adaptation Design Patterns

Name	Category	Description	Selected Sources
Sensor Factory	M, C	Deploy sensors across a distributed infrastructure and probe components.	[3, 7, 9, 24]
Reflective Monitoring	M, S	Perform introspection on a component and dynamically alter a sensor's behavior.	[1, 5]
Content-based Routing	M, S	Route monitoring information based on the content of the message.	[9, 13, 30]
Case-based Reasoning	DM, S	Rule-based approach to selecting a reconfiguration plan.	[9, 15]
Divide & Conquer	DM, S	Systematically decompose a complex reconfiguration plan into simpler reconfiguration plans.	[9, 12]
Adaptation Detector	DM, S	Interpret monitoring data and determine when an adaptation is required.	[15, 20]
Architectural-based	DM, S	Provide an architecture-based approach for selecting reconfiguration plans.	[3, 23]
Tradeoff-based	DM, S	Systematically select a reconfiguration plan that best balances multiple objectives.	[3, 23]
Component Insertion	R, S	Safely insert and initialize a component at run time.	[3, 10, 19]
Component Removal	R, S	Safely remove a component at run time.	[3, 10, 19]
Server Reconfiguration	R, B	Safely reconfigure a server - client component architecture at run time.	[3, 10, 17]
Decentralized	R, B	Safely insert and remove components from a decentralized component architecture at run time.	[10, 19]

- Pattern Name: The pattern name uniquely identifies and describes the pattern.

  -Sensor Factory.

- Classification: The classification facilitates the organization of patterns based on the purpose of the pattern.

  -Monitoring, Creational.

- Intent: A brief description of the problem(s) that the pattern addresses.

  -Systematically deploy software sensors across a network to probe distributed components.

- Context: Describe the conditions and context in which the pattern may be applied.

  -The Sensor Factory pattern may be used when components to be monitored are distributed and each component provides an interface that can be probed for the required information.

- Motivation: A description of sample goals and objectives of a system that motivate the use of the pattern. Use-cases and use-case diagrams describe goals of the pattern application.

  -External monitoring mechanisms must effectively collect information about the running system to properly evaluate a system's operational status. The objective of the Sensor Factory design pattern is to manage distributed sensors across a networked environment such that they may probe distributed components. The Sensor Factory design pattern captures the structural relationship between sensors, clients, and components. By decoupling sensors from clients and components, the monitoring infrastructure is flexible and more amenable to change.

- Structure: A representation of the classes and their relationships depicted in terms of UML class diagrams.

  -A UML class diagram for the Sensor Factory pattern can be found in Figure 2.

  Two types of sensors can be found in this pattern. *SimpleSensors* can handle booleans, integers and real data types. *CmoplexSensors* can report more complex data types and aggregate the outputs of *SimpleSensors*. Both *SimpleSensor* and *ComplexSensor* inherit from the *AbstractSensor* abstract class, thereby providing an interface with basic functionality such as pushing or polling for data.

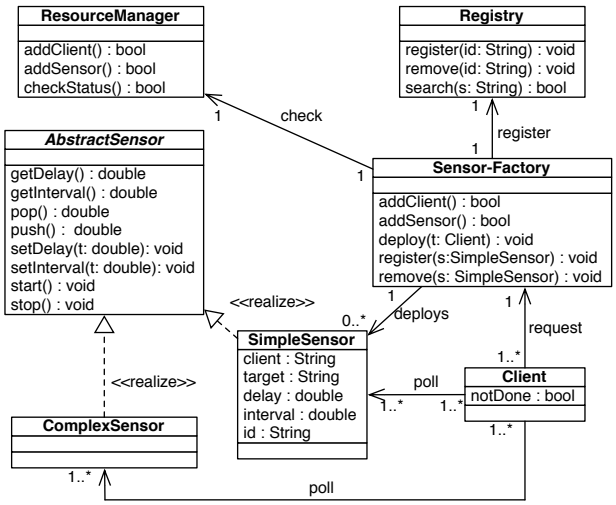

Figure 2: UML class diagram of the SensorFactory Pattern

- Participants: Itemizes the classes and objects that are included in the adaptation design pattern and lists their responsibilities.

  -**Abstract-Sensor**: *SimpleSensor* and *ComplexSensor* both inherit from this abstract class. As a result, these sensors share an interface to common operations such as pushing and pulling data.

  -**Client**: This class is used to represent any component that needs to perform either internal or external monitoring.

  -**ComplexSensor**: This type of sensor comprises more computing resources on-board than a *SimpleSensor* does. As a result, a *ComplexSensor* is capable of reporting complex data types, aggregating various *SimpleSensor* data feeds, and performing on-board computations.

  -**Registry**: This class is responsible for tracking deployed sensors across the network. Each entry should at least record the sensor name, the sensor type, the *Client* to which it is providing data, and the component it is monitoring. Additionally, this class provides a search functionality based on

the available fields.

-**ResourceManager**: This class has two responsibilities. First, it determines if an existing sensor can be shared with one or more clients. A sensor can be shared as long as it does not violate any existing constraint. Second, it determines if the system has enough resources to deploy a new sensor across the network.

-**SensorFactory**: *Clients* must interact with this class in order to gain access to a sensor. It regulates the dynamic access and management of sensors across a network.

-**SimpleSensor**: The most basic sensor available. It is capable of reporting boolean, integer, and real data types. Additionally, it can be configured to poll a component at different intervals and periods.

- Behavior: Provides an illustrative representation of scenarios for class and object interaction. Also gives a description of the behavior of the pattern by using sample or high-level, abstract UML state and sequence diagrams.

  -Figure 3 shows a UML sequence diagram for an example of the Sensor Factory pattern in a distributed monitoring system. The *Client* requests a *SimpleSensor* (an active networked sensor) from the *SensorFactory*. The *SensorFactory* determines whether an existing *SimpleSensor* already provides the desired information. If not, then *SensorFactory* prompts the *ResourceManager* to determine if another *SimpleSensor* can be deployed across the network without violating any quality of service constraints. If so, then *SensorFactory* instantiates a new *SimpleSensor* and initializes it to some default sensor setting. *SensorFactor* then notifies the *Client* that the *SimpleSensor* is ready for use. *Client* polls the *SimpleSensor* until it is done monitoring.

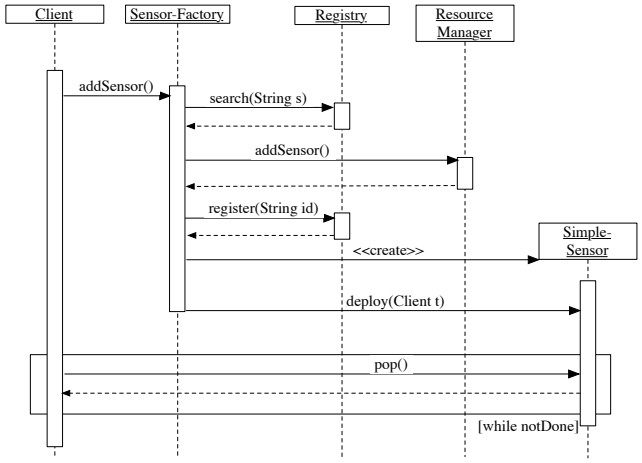

Figure 3: UML sequence diagram example of the Sensor Factory Pattern

- Consequences: Describes how objectives are supported by a given pattern and gives the trade-offs and outcomes of the pattern application.

  -This design pattern reuses the provided functionality and interface of a distributed component to extract the desired attributes. However, if a component's interface is excessively polled, then it could interfere and alter the component's behavior.

  -Different types of sensors can be systematically deployed at run time while providing a flexible monitoring infrastructure that is amenable to adaptation.

  -This design pattern ensures system integrity by accessing a component's attributes through its interface.

  - To avoid wasting computational resources, *Registry* and *ResourceManager* share existing sensors whenever possible.

  - This design pattern introduces a management layer between a *Client* and a sensor. This additional overhead may degrade performance.

  - Monitoring is supported only for components that provide an interface to the required attributes.

- Constraints: Contains LTL and A-LTL templates and a prose description of the constraints that must be satisfied by a given design pattern application.

  -Property1: Omitted here, but discussed in case study.

  -Property 2: Globally, it is always the case that if *Client* requests a sensor to *Sensor-Factory*, then *Sensor-Factory* will eventually grant access to a sensor.

  $\Box$ ((*Client*.request(sensor)) $\rightarrow$
  $\Diamond$ (*Sensor-Factory*.grant(*Client*))

  This liveness property guarantees that a *Client* will eventually get access to a sensor.

- Related Patterns: Additional design patterns that are commonly used in conjunction.

  -Adapter Design Pattern [8]: This pattern can enable the interaction between a *Client* and a sensor whenever their interfaces are incompatible.

  -Reflective Monitoring Design Pattern [25]: This pattern can be used whenever a component does not provide an interface to the required attributes. Such values may be accessible through Introspection.

  -Adaptation Detector Design Pattern [25]: This pattern is responsible for interpreting the results provided by a sensor and determining when an adaptation is required.

- Known Uses: Lists the sources used to harvest the design pattern.

  -REsource MOnitoring for network-aware applicationS [6], SNMP4J-Agent [7], Rainbow Adaptation Framework [3, 9], A Distributed Monitoring Service Architecture (MonALISA) - via SNMP [24].

5. EXAMPLE INSTANTIATION

This section presents a proof of concept study in which we re-engineered an adaptive application from scratch using our adaptation design patterns. We first describe the adaptive application and its functional and adaptive requirements. We then overview our pattern-based design and compare it against the original framework-based design and implementation.

5.1 Application Description

The Z.com case study was originally developed using the Rainbow framework [3]. Z.com is a fictional news site that uses adaptation to address the "slashdotting effect" where news sites that are widely publicized are unable to handle the large number of content requests, and they either suffer from high latency or are unable to serve content alto-

gether. Z.com was modeled as a set of clients and servers with the overall constraint that latency must fall within a given threshold. Nonetheless, adaptation concerns for Z.com are multi-faceted; some of the utility concerns that must be balanced include cost, latency, and fidelity.

This study uses several of the adaptation design patterns presented in this paper to re-engineer the Z.com adaptive web server [3]; to distinguish the two designs, we call the pattern-based system ZAP.com. Although ZAP.com is implemented through a complementary approach, it exhibits the same functionality and observable behavior to Z.com, built with the Rainbow framework. Specifically, ZAP.com is able to handle the same reconfiguration scenarios as Z.com. Having two implementations of the same adaptive system enables us to perform a more comprehensive comparison of the key differences between the two adaptive design approaches.

5.2 Application Requirements

The same functional and adaptive requirements of Z.com apply to ZAP.com. Specifically, the following requirements were previously identified by Garlan *et al.* for Z.com [3]:

1. The news server will provide basic HTML functionality to requesting clients.
2. The operational cost may not be exceeded at any time.
3. The quality of the content should be the best one possible. Specifically, whenever possible, service client's requests in graphical content mode.
4. The system will avoid losing customers due to high response times if it can somehow provide faster content. Specifically, if the server's average response time is too high, then the content may be switched to textual mode in order to avoid the costs of transmitting large files.

Given the objective of minimizing operational costs and latency, while providing graphical news content whenever possible, Garlan *et al.* reasoned about the possible adaptation scenarios that might arise for Z.com. For instance, Z.com will increment its server pool by one integral amount if the response time is high and the budget will not be exceeded. Otherwise, Z.com will switch to textual content mode if it is not already in that mode. Additionally, when the response time is low, Z.com will decrement its server pool size by one integral amount if it is near budget limit. If the response time is low, then the servers will be switched to graphical mode if they are not already in that mode. Lastly, when the response time is in the medium range, Z.com will switch to graphical mode if the mode is textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

5.3 Application Design

We re-engineered the Z.com application in three major stages to obtain ZAP.com, closely following the model-based development process for adaptive systems [27, 32]. First, we modeled and implemented the business logic according to the local properties and functional requirements identified for Z.com [3]. Next, based on the possible adaptation scenarios, we identified and instantiated a set of monitoring and decision-making design patterns that were applicable to ZAP.com. To ensure our design satisfied functional properties, we analyzed the resulting models against local properties and invariants before we implemented them. Lastly, we modeled and implemented the adaptive logic responsible for reconfiguring the server architecture and integrated it with the rest of the system.

Functional Logic. ZAP.com is modeled as an object-oriented multi-threaded server-client architecture. Specifically, the ZAP.com design comprises a set of classes that accept incoming HTML requests from different web browsers, determines the current system workload, and redirects requests in order to balance the overall system workload. The design models were implemented in the Java programming language. As such, the ZAP.com implementation is able to service common HTML requests such as retrieving files across a network.

Monitoring and Decision-Making Logic. We selected a set of monitoring and decision-making patterns based on the context of the Z.com application. Specifically, we applied the Sensor Factory pattern to periodically monitor the average latency of Servers for two reasons. First, a distributed monitoring scheme is required for Z.com's networked architecture. Second, our functional logic already provides an interface to the attributes that need to be monitored. Garlan *et al.* followed a similar approach based on the observation that system administrators would have access to monitoring information from the application's interface [3]. As Figure 4 illustrates, Rainbow uses a GaugeCoordinator to deploy and manage RegularPatternGauges, which derive their interface from the AbstractGauge interface. This subset of Rainbow's monitoring infrastructure is similar in purpose, structure, and behavior to the *Sensor Factory* pattern.

We used Hydra [22] to automatically translate the state-based models of ResourceManager, SensorFactory, and SimpleSensor into Promela code. A constraint violation was found when we attempted to verify the Promela models in the SPIN model checker [14]. The violated property stated that if ResourceManager denies a Sensor request, then SensorFactory will not deploy that Sensor. Specifically,

$$\Box\ (\text{(ResourceManager.deny(Client)} \rightarrow \neg\ \text{Sensor-Factory.createSensor(Client))}$$

The instantiated *Sensor Factory* pattern allowed the existence of multiple ResourceManagers. Although this did not seem problematic at first glance, it enabled the following scenario: "the same Sensor request is denied by instance x of ResourceManager and granted by instance y of ResourceManager." As a result, the Sensor request is granted even though an existing ResourceManager explicitly denied the request. To ensure consistency within the monitoring infrastructure and the deployed Sensors, we revised our instantiated models of the *Sensor Factory* pattern to allow only one active ResourceManager at any time.

We also applied two decision-making patterns to ZAP.com, *Adaptation Detector* (typically useful for most adaptive systems) and *Case-based Reasoning*. The *Adaptation Detector* pattern was selected to process the monitoring data supplied by SimpleSensors and detect when a reconfiguration was required. Specifically, a HealthIndicator compared values reported by SimpleSensors with the values stored in Threshold. If a Threshold was exceeded, then HealthIndicator would create a Trigger and send it to the InferenceEngine. As Figure 4 illustrates, Rainbow uses a similar approach to detect when a reconfiguration is needed. More specifically, Rainbow uses

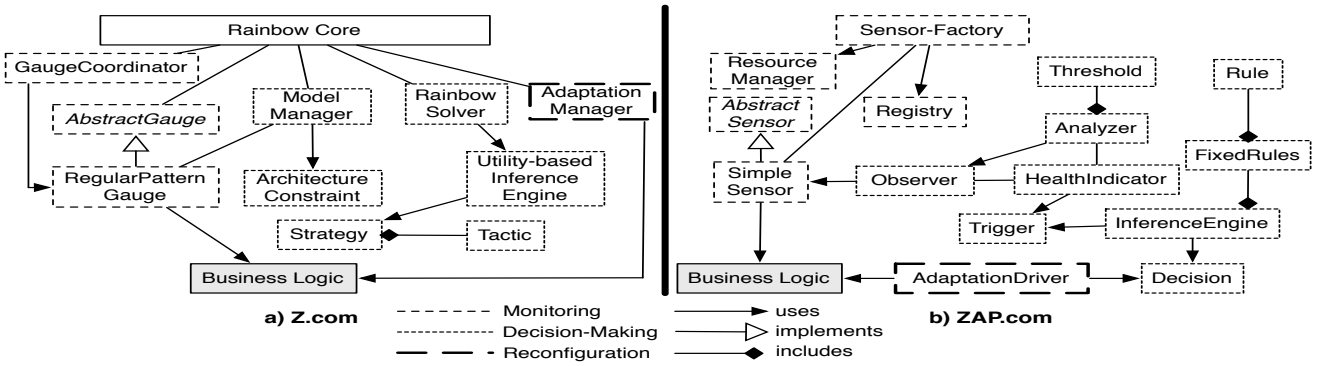

Figure 4: Elided structural models for framework-built Z.com and pattern-based ZAP.com

a ModelManager and an ArchitectureConstraint to map monitoring data against specific constraints in the application.

As with the *Sensor Factory* pattern, we used Hydra to automatically translate UML state-based models of HealthIndicator, Analyzer, and Observer into Promela code. We then used the SPIN model checker to analyze the two properties specified for the *Adaptation Detector* design pattern. First, we wanted to ensure that if Observer received any monitoring value, then it would eventually be compared against its associated Threshold. Specifically,

□ (Observer.getData(Sensor) →
◊ Analyzer.compare(Data,Threshold))

Likewise, we wanted to ensure that if a Threshold was exceeded, then a Trigger for adaptation would be created. Specifically,

□ (Analyzer.compare(Data).'True' →
◊ HealthIndicator.send(Trigger))

The SPIN model checker did not find any violations for these properties in the instantiated models.

The *Case-based Reasoning* pattern was chosen to select a reconfiguration plan based on the available monitoring information. Specifically, the InferenceEngine attempts to match received Triggers against a set of Rules (see Figure 4). If a Trigger matches a Rule, then the associated reconfiguration plan is encapsulated in a Decision and sent to the reconfiguration infrastructure. In contrast, Rainbow uses a utility-based approach for selecting reconfiguration plans [3]. In Rainbow, reconfiguration steps are encapsulated into Tactics, and sets of Tactics are further encapsulated into Strategies, which form actual reconfiguration plans. Each Strategy is analyzed by the Utility-based InferenceEngine to select the reconfiguration plan that will best balance multiple objectives across the system. Although the decision-making logic in ZAP.com is comparatively simpler, we were able to encode reconfiguration plans as a set of "if-then" rules based on the adaptation analysis performed by Garlan *et al.* [3].

As with the previous instantiated design patterns, we used Hydra to translate UML state-based models of InferenceEngine and FixedRules into Promela code. These Promela models were then analyzed with the SPIN model checker against two properties. First, we wanted to ensure that if a Trigger is received, then it is always the case that a Decision is produced. Specifically,

□ (InferenceEngine.trigger(Trigger) →
◊ InferenceEngine.action())

The ZAP.com design includes a default rule and decision pair to enforce this constraint in the event that no other pair matches at run time.

Second, we wanted to ensure that if a Decision is produced, then it is always the case that it is logged. Specifically,

□ (InferenceEngine.action() →
◊ Log.log(Trigger,Rule,Decision))

Logging the reconfiguration Decision and why it was selected will help developers understand how their system is behaving at run time. Again, the SPIN model checker did not find any property violations for either of these two properties.

Reconfiguration Logic. At this stage we have designed the ZAP.com application to support monitoring and decision-making capabilities. Specifically, we can periodically observe the average latency, detect when a reconfiguration is required, and select a reconfiguration plan that will yield the desired behavior. Garlan *et al.* identified four possible reconfigurations for the Z.com adaptive news server [3]. Two of those reconfigurations involve tuning parameters to alternate between content delivery modes. The two other reconfigurations involve either adding or removing a Server at run time. In contrast to the parameter reconfiguration, the adaptive infrastructure must first prepare the system before a Server is added or removed at run time. That is, the Server must be guided towards a quiescent state to guarantee that the reconfiguration process will not leave the system in an inconsistent state. For Z.com, incoming client requests need to be queued so they may be processed after the reconfiguration is complete. Otherwise, client requests may be lost as the system is reconfigured.

We used the *Server Reconfiguration* pattern to safely reconfigure the ZAP.com application in scenarios that involved the addition or removal of a Server. The *Server Reconfiguration* pattern, in turn, reuses the *Component Insertion* and *Component Removal* reconfiguration design patterns to safely add and remove components, respectively. For instance, to add a Server at run time, the AdaptationDriver first loads and initializes a new Server and LoadBalancer (from the business logic). The AdaptationDriver then inserts a Request Buffer to store incoming requests during the reconfiguration procedure. Then the AdaptationDriver sends passivate commands to both the Servers and LoadBalancer so they can be safely reconfigured. Once these components are *passive*, the LoadBalancer can be driven to a *quiescent* state so it can be removed from the system. Notifications are then sent by

the AdaptationDriver to *activate* the affected components. Finally, once all the queued requests are serviced, the reconfiguration is complete and the system continues to operate as normal.

At this stage, the ZAP.com application comprises instantiations of one monitoring pattern, two decision-making patterns, and three reconfiguration patterns. This set of six adaptation design patterns realizes our ZAP.com application with self-adaptive behavior. That is, sensors periodically probe the Servers for the average latency, and whenever substantial change is detected, an adaptation request is issued. The decision-making determines which reconfiguration plan to apply based on the monitoring information. The reconfiguration plan is then applied by the AdaptationDriver. Specifically, the AdaptationDriver either switches the content delivery mode to textual/graphical or adds/removes a server at run time.

When compared to the Rainbow version of Z.com, our application provides the same observable functionality and reconfiguration capabilities. For validation purposes, we selected design patterns that most closely captured the design strategy used for Z.com. We note that in several cases, different design patterns could have been used to account for new monitoring and decision-making technology. Next we give comparisons of framework versus design pattern-based implementations.

6. DISCUSSION

Re-engineering the Z.com application originally presented by Garlan *et al.* in [3] enables us to compare and contrast the advantages and disadvantages of both approaches. Our design pattern approach has several advantages over a framework-oriented approach at developing dynamically adaptive systems. For instance, our design patterns provide general models that need to be instantiated and customized before they are implemented. Since models operate at a higher-level of abstraction than frameworks, they impose fewer initial constraints upon the system being developed. Frameworks, on the other hand, incorporate many design decisions already made by the framework developers, which may or may not be compatible with the application's needs and requirements. Likewise, design patterns do not entail a steep learning curve in order to apply them successfully. To properly use a framework, however, a developer must understand the underlying framework mechanisms and how they relate to the application being built. In particular, developers must determine how their application will be reconfigured by the adaptation framework at run time to ensure proper system behavior. Additionally, instantiated versions of our design patterns can be analyzed through formal verification tools and techniques to ensure a design satisfies certain key properties before it is implemented. Unless verification capabilities were built into the design of a framework, attempting to verify its correctness is, at best, impractical. Lastly, with our design pattern approach, developers select only those adaptation mechanisms their application will require. In contrast, adaptation-oriented frameworks provide infrastructure to perform monitoring, decision-making, and reconfiguration tasks for a wide range of applications within a domain; the overall infrastructure is needed for the adaptive application, even if not all the features are needed or used.

Framework-oriented approaches, on the other hand, have several notable advantages over our design patterns for building dynamically adaptive systems. For instance, adaptation-enabling frameworks provide large amounts of code that can be directly reused when building adaptive applications. If the application and the framework share the same context and domain, then a large portion of development overhead can be avoided by reusing the framework's code. Design patterns, however, offer no code at all. After the system is designed and the patterns are instantiated, they must be implemented by developers. Likewise, adaptation-enabling frameworks, such as Rainbow [9], tend to support a wide range of adaptation mechanisms and techniques. With a pattern-based approach, each desired functionality must be carefully implemented and integrated with the application. Lastly, with a pattern-based approach, developers are exposed to some details and complexities of reconfiguring applications at run time. Specifically, developers must both identify the quiescent states that are safe for reconfiguring an application as well as implement the mechanisms for guiding the system towards those states. In contrast, adaptation-enabling frameworks largely hide the internals of dealing with specific reconfiguration scenarios from developers. As a result, developers have limited exposure to the details and complexities of reconfiguring applications at run time.

In general, developers should leverage adaptation-oriented frameworks when the application's context is compatible with the framework's services. If the application's context and adaptation needs differ significantly from the existing adaptation-oriented framework, then our adaptation-oriented patterns may provide a more convenient development alternative. Moreover, it may be possible to integrate our adaptation-oriented patterns with existing adaptation-oriented frameworks depending on the design of the framework.

7. CONCLUSIONS

This paper introduced twelve adaptation-oriented design patterns to support monitoring, decision-making, and reconfiguration of adaptive systems where, the patterns facilitate the separate development of the functional logic and the adaptive logic. Each design pattern is the product of studying at least two candidate successful design solutions and generalizing them so that they may be applied across different domains. We extended the pattern template used by Gamma *et al.* [8] for describing design patterns with *Behavioral* and *Constraints* fields. The information provided in the template enables developers to understand the consequences and trade-offs incurred by applying a pattern. Furthermore, the use of a design pattern template enforces the uniform organization of every adaptation design pattern, thus facilitating their use. For each pattern, we have also identified sets of related patterns that are frequently applied together in practice. To assess their maturity, we validated each design pattern against instances in adaptive systems that were not used in the harvesting process. In addition, we successfully applied a subset of the patterns in the development of an adaptive web server. This example helped illustrate how our adaptation-focused design patterns integrate with the model-based development process [27, 32] for adaptive systems to construct self-adaptive and autonomic computing systems.

Several directions for future work are possible. First, additional design patterns for both domain-specific and domain-

independent adaptation could be identified and integrated with the set of design patterns presented in this paper [26]. Second, we are examining how these design patterns can be inserted into a non-adaptive application through the use of aspect-oriented techniques [29]. Lastly, we are exploring the use of digital evolution techniques to determine how adaptation design patterns can be automatically instantiated and integrated into legacy systems to meet adaptation needs.

8. REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[2] J. Cámara, C. Canal, J. Cubo, and J. M. Murillo. An aspect-oriented adaptation framework for dynamic component evolution. *Electron. Notes Theor. Comput. Sci.*, 189:21–34, 2007.

[3] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems*, pages 2–8, New York, NY, USA, 2006. ACM.

[4] G. Coulson, P. Grace, G. Blair, W. Cai, C. Cooper, D. Duce, L. Mathy, W. K. Yeung, B. Porter, M. Sagar, and W. Li. A component-based middleware framework for configurable and reconfigurable grid computing. *Concurr. Comput. : Pract. Exper.*, 18(8):865–874, 2006.

[5] D. Dawson, R. Desmarais, H. M. Kienle, and H. A. Muller. Monitoring in adaptive systems using reflection. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 81–88, Leipzig, Germany, 2008. ACM.

[6] T. Dewitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. Remos: A resource monitoring system for network aware applications, 1997.

[7] F. Fock. The SNMP API for java. http://www.snmp4j.org/index.html.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 195.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[10] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *WICSA'04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 79, Washington, DC, USA, 2004.

[11] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes. Dynamic reconfiguration in sensor middleware. In *MidSens'06: Proc. of the Intl. workshop on Middleware for sensor networks*, pages 1–6, New York, NY, USA, 2006. ACM.

[12] M. Hans. The control architecture of care-o-bot ii. *E. Prassler et al (Eds.): Advances in Human-Robot Interaction*, pages 321–330, 2004.

[13] D. Heimbigner and A. Wolf. Definition, deployment and use of gauges to manage reconfigurable component-based system. Technical Report A082924, University of Colorado, 2004.

[14] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[15] G. Kaiser, P. Gross, G. Kc, and J. Parekh. An approach to autonomizing legacy systems. In *Proceedings of the first Workshop on Self-Healing, Adaptive, and Self-MANaged Systems*, 2002.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[17] T. Kindberg. Reconfiguring client-server systems. In *International Workshop on Configurable Distributed Systems*, 1993.

[18] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, C. Magalh, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware'00: IFIP/ACM Intl. Conf. on Dist. Sys. Platforms*, pages 121–143. Springer-Verlag, 2000.

[19] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. on Soft. Eng.*, 16(11):1293–1306, 1990.

[20] A. Lau. Design patterns for software health monitoring. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computing Systems*, pages 467–476, Washington, DC, USA, 2005. IEEE Computer Society.

[21] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[22] W. E. McUmber and Betty H.C. Cheng. A general framework for formalizing uml with formal languages. In *ICSE'01: Proc. of the 23rd Intl. Conf. on Soft. Eng.*, pages 433–422, Washington, DC, USA, 2001. IEEE Computer Society Press.

[23] M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G. A. Papadopoulos, and A. Chimaris. Distributed context management in a mobility and adaptation enabling middleware (madam). In *SAC'06: Proc. of the 2006 ACM symposium on Applied Computing*, pages 733–734, New York, NY, USA, 2006. ACM.

[24] H. Newman, I. Legrand, P. Galvez, R. Voicu, and C. Cistoiu. MonALISA: A Distributed Monitoring Service Architecture. In *Proceedings of the 2003 Conference for Computing in High Energy and Nuclear Physics*, March 2003.

[25] A. J. Ramirez. Design patterns for developing dynamically adaptive systems. Master's thesis, Michigan State University, East Lansing, MI 48823, 2008.

[26] A. J. Ramirez and Betty H.C. Cheng. Design patterns for monitoring adaptive uls systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software Systems-Intensive Systems*, pages 69–72, New York, NY, USA, 2008. ACM.

[27] A. J. Ramirez and Betty H.C. Cheng. Verifying and analyzing adaptive logic through uml state models. In *Proc. of the 2008 IEEE Intl. Conf. on Soft. Testing, Verification, and Validation*, pages 529–532, Lillehammer, Norway, 2008.

[28] S. M. Sadjadi and P. K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 74–83, 2004.

[29] S. M. Sadjadi, P. K. McKinley, and Betty H.C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *DEAS'05: Proc. of the 2005 workshop on Design and Evolution of Autonomic Application Software*, New York, NY, USA, 2005. ACM.

[30] A. Zeidler and L. Fiege. Mobility support with REBECA. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003. IEEE Computer Society.

[31] J. Zhang and Betty H.C. Cheng. Specifying adaptation semantics. In *WADS'05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM.

[32] J. Zhang and Betty H.C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[33] J. Zhang, H. J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the Eighth International Conference on Aspect-Oriented Software Development*, 2009.