Software Architecture-Based Self-Adaptation

David Garlan, Bradley Schmerl, and Shang-Wen Cheng

Abstract Increasingly, systems must have the ability to self-adapt to meet changes in their execution environment. Unfortunately, existing solutions require human oversight, or are limited in the kinds of systems and the set of quality-of-service concerns they address. Our approach, embodied in a system called Rainbow, uses software architecture models and architectural styles to overcome existing limitations. It provides an engineering approach and a framework of mechanisms to monitor a target system and its environment, reflect observations into a system's architecture model, detect opportunities for improvement, select a course of action, and effect changes in a closed loop. The framework provides general and reusable infrastructures with well-defined customization points, allowing engineers to systematically customize Rainbow to particular systems and concerns.

1 Introduction

Imagine a world where a software engineer could take an existing software system and specify an objective, conditions for change, and strategies for adaptation to make that system self-adaptive where it was not before. Furthermore, imagine that this could be done in a few weeks of effort and be sensitive to maintaining business goals and other properties of interest. For example, an engineer might take an existing client–server system and make it self-adaptive with respect to a specific performance concern such as high latency. He might specify an objective to maintain request-response latency below some threshold, a condition to change the system if the latency rises above the threshold, and a few strategies to adapt the system to fix the high-latency situation. Another engineer might make a coalition-of-services system self-adaptive to network performance fluctuations, while limiting cost of operating the infrastructure. Still another engineer might make a cluster of servers self-adaptive to certain security attacks.

D. Garlan (✉)
Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA
e-mail: garlan@cs.cmu.edu

M.K. Denko et al. (eds.), *Autonomic Computing and Networking*,
DOI 10.1007/978-0-387-89828-5_2,

Today, when increasingly systems have the requirement to self-adapt with minimal human oversight, it is becoming necessary to meet this vision. Systems must cope with variable resources, system errors, and changing user priorities, while maintaining, as best they can, the goals and properties envisioned by the engineers and expected from the users. Software engineers lack the tools and techniques to engineer a system with self-adaptation.

Engineers and researchers alike have responded to and met this self-adaptation need in somewhat limited forms through programming language features such as exceptions and in algorithms such as fault-tolerant protocols. But these mechanisms are often specific to the application, tightly bound to the code, and usually provide only localized treatment of system errors. As a result, self-adaptation for today's systems are costly to build, often taking many man-months to retrofit systems.

In contrast, the vision outlined above requires an approach that makes it possible for engineers to easily define adaptation policies that are global in nature, and that take into consideration business goals and quality attributes. In particular, we require that engineers be able to augment existing systems to be self-adaptive without rewriting them from scratch, that self-adaptation policies and strategies can be reused across similar systems, that multiple sources of adaptation expertise can be synergistically combined, and that all of this can be done in ways that support maintainability, evolution, and analysis.

In this chapter, we describe an approach to achieving these goals using architecture-based self-adaptation techniques. In particular, our approach abstracts observed behavior of an executing system into properties of an architectural model, where they can be reasoned about using a variety of existing architectural analysis techniques. The results of these analyses can then be used to reason about changes that should be made to a system to improve or correct the system's achievement of the quality attributes.

Our approach is embodied in a system called Rainbow, which focuses on two challenges to achieve cost-effective self-adaptation: (1) an approach and mechanism that reduces engineering effort and (2) representation of adaptation knowledge. Rainbow provides an engineering approach and a framework of mechanisms to monitor a system and its executing environment, reflect observations into an architectural model of the system, determine any problem states, select a course of action, and effect changes. By leveraging the notion of architectural style to exploit commonality of systems, the framework provides a general and reusable infrastructure with well-defined customization points to cater to a wide range of systems. The framework also provides a set of abstractions that allow engineers to focus on adaptation concerns, facilitating an adaptation engineering workflow for the systematic customization of Rainbow. To emulate the mundane and routine adaptation tasks performed by system administrators, Rainbow provides a language, called Stitch, to represent the adaptation techniques using first-class adaptation concepts. It offers modularity with respect to quality dimension and domain expertise, strategies with condition and effect, a mechanism to tailor to particular styles, and the use of utility theory to compute the best adaptation path under uncertainty.

In this chapter, we introduce the ideas behind architecture-based self-adapting systems; briefly survey the research landscape; discuss the research and engineering challenges, particularly with respect to autonomic behavior for distributed, networked systems; and describe the Rainbow approach and how it addresses these challenges. We also give examples of its use in the context of autonomic networks, focusing on adaptations to improve qualities such as fidelity, performance, security, and cost of operation.

2 Overview of Autonomic and Self-Adaptive Systems

Overcoming the challenges of self-adaptation and allowing managed systems to self-adapt with minimal human oversight requires closing the "loop of control." Software systems have traditionally been designed as *open-loop* systems: once a system is designed for a certain function and deployed, its extra-functional quality attributes typically remain relatively unchanged. In most cases, if something goes wrong, humans must intervene, often by restarting the failed subsystem or taking the entire system offline for repair. This results in high costs in system downtime, personnel costs, and decreased revenue through system unavailability.

To address this problem, a number of researchers have proposed an alternative approach that uses external software mechanisms to maintain a form of closed-loop control over the target system (e.g., [26, 30, 39]). Such mechanisms allow a system to self-adapt dynamically, with reduced human oversight. Minimally, closed-loop control consists of mechanisms that monitor the system, reflect on observations for problems, and control the system to maintain it within acceptable bounds of behavior. This kind of system is known as a *feedback control system* in control theory [42].

Feedback control systems have typically been applied to control physical systems. For simple systems, the control model may be built-in to the design. For example, a home thermostat that measures room temperature and checks it against the set point, controlling a home heating and cooling system, will typically have a simple built-in thermodynamic model. In more complex systems an explicit process model is necessary for effective control [42]. For example, an air conditioning system for a large building that monitors and controls multiple locations would require an explicit model of the building partitions and temperatures to efficiently control which cooling units to turn on and when.

For software systems, the external controller requires an explicit model of the target system in order to reflect on observations and to configure and repair the system [39]. Monitoring mechanisms extract and aggregate target system information to update the model. An evaluation mechanism detects problems in the target system as reflected in the model. The appearance of a problem triggers an adaptation mechanism to use the model to determine a course of action. The mechanism then propagates the necessary changes to the target system to fix the problem.

In principle, external mechanisms have a number of benefits over internal mechanisms. External control separates the concerns of system functionality from those of

adaptation (or "exceptional") behaviors. With the adaptation mechanism as a separate entity, engineers can more easily modify and extend it, and reason about its adaptation logic. Furthermore, the separation of mechanisms allows the application of this technique even to legacy systems with inaccessible source code, as long as the target system provides, or can be instrumented to provide, hooks to extract system information and to make changes. Finally, providing external control with generic but customizable mechanisms (e.g., model management, problem detection, strategy selection) facilitates reuse across systems, reducing the cost of developing new self-adaptive systems.

2.1 The IBM Autonomic Framework

The IBM Autonomic Computing Initiative codified an external, feedback control approach in its Autonomic Monitor-Analyze-Plan-Execute (MAPE) Model [28]. Figure 1 illustrates the MAPE loop, which distinguishes between the *autonomic manager* (embodied in the large rounded rectangle) and the *managed element*, which is either an entire system or a component within a larger system. The MAPE loop highlights four essential aspects of self-adaptation:

1. **Monitor**: The monitoring phase is concerned with extracting information—properties or states—out of the managed element. Mechanisms range from source-code instrumentation to non-intrusive communication interception.
2. **Analyze**: is concerned with determining if something has gone awry in the system, usually because a system property exhibits a value outside of expected bounds, or has a degrading trend.
3. **Plan**: is concerned with determining a course of action to adapt the managed element once a problem is detected.
4. **Execute**: is concerned with carrying out a chosen course of action and effecting the changes in the system.

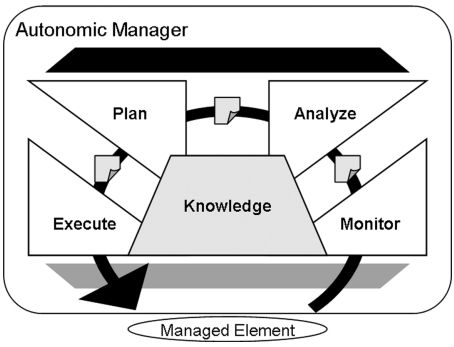

Fig. 1 The IBM Autonomic MAPE Reference Model

Shared between these four phases is the **Knowledge** component, which contains models, data, and plans or scripts to enable separation of adaptation responsibilities and coordination of adaptations. The Rainbow framework provides components that fulfill each of these four phases and the *knowledge* to support self-adaptation.

3 Software Architecture and Architecture-Based Self-Adaptation

A key issue in using an external model is to determine the appropriate kind of models to use for software-based systems. Each type of model has certain advantages in terms of the analyses and kinds of adaptation it supports. In principle, a model should be abstract enough to allow straightforward detection of problems in the target system, but should provide enough fidelity to determine remedial actions to take to fix the problem. State machines, queuing theory, graph theory, differential equations, and other mathematical models [40, 42] have all been used for model-based, external adaptation of software systems.

We, among others, use a system's software architecture as the external model for dynamic adaptation [19, 39]. The architecture of a software system is an abstract representation of the system as a composition of computational elements and their interconnections [44]. Specifically, an *architecture model* represents the system as a graph of interacting components.[1] Nodes in the graph, termed *components*, represent the principal computational elements and data stores of the system: clients, servers, databases, user interfaces, etc. Arcs, termed *connectors*, represent the pathways of interaction between the components. This is the core architectural representation scheme adopted by a number of architecture description languages (ADLs), such as Acme [20] and xADL [13].

The use of software architecture as the basis for self-adaptation, termed *architecture-based self-adaptation*, holds a number of potential promises. A rich body of work on architecture trade-off analysis techniques used at system design time facilitates runtime self-adaptation. As an abstract model, an architecture model provides a global perspective on the system and exposes the important system-level behaviors and properties. As a locus of high-level system design decisions, the model makes system integrity constraints explicit, thereby helping to ensure the validity of a change. For example, the architecture model can expose important properties such as throughput and bandwidth, allowing the overall throughput or performance of the system to be analyzed. Furthermore, the model might be associated with explicit constraints on the architecture that, for example, forbid cycles. This knowledge can be used at runtime to reason about the effect of a change on the system's throughput or structure. See [18] for a discussion of this concept for performance evaluation.

---

[1] We are primarily interested in the component–connector view [11] because it characterizes the abstract state and behavior of the system at runtime to enable reasoning about problems and courses of adaptation.

Crucial for architecture-based self-adaptation is the choice of the *architectural style* used to represent the target system. A style (e.g., pipe-filter) provides the vocabulary to describe the architecture of a system in terms of a set of component types (e.g., filter) and connector types (e.g., pipe), along with the rules for composition (e.g., no cycles) [1]. A style might also prescribe the properties associated with particular element types (e.g., throughput on a pipe). Usually associated with a style is a set of analytical methods to reason about properties of systems in that style. For example, systems in the MetaH style supports real-time schedulability analysis [16].

For self-adaptation, given some quality objectives, each style may guide the choice of system properties to monitor, help identify strategic points for system observation, and suggest possible adaptations. To illustrate this, consider a signal-processing system with an architecture in the pipe-filter style. This style constrains the system to a data-flow computation pattern, points to throughput as a system property, identifies the filter as a strategic point for measuring throughput, and suggests throughput analysis for reasoning about overall system throughput. The pipe-filter style may suggest adaptations that swap in variants of filters to adjust throughput, create redundant paths to improve reliability, or add encryption to enhance security. In contrast, consider a different system in the client–server style. This style highlights request-response latency as a key property, identifies the client as a strategic point for measuring latency and the server for load, and suggests the use of queuing theory to reason about service time and latency. The style may suggest an adaptation that switches clients to less loaded servers to reduce latency.

4 Related Work

To date, several dynamic software architectures and architecture-based adaptation frameworks have been proposed and developed [7, 24, 39], including an effort to characterize the style requirements of *self-healing systems* [35]. Below, we examine a representative set of approaches, categorizing each by its primary focus, then highlighting its main features. Broadly speaking, related approaches focus on formalism and modeling, or mechanisms of adaptation. A third category addresses distribution and decentralization of control.

4.1 Distributed, Decentralized Adaptation

Work on self-organizing systems in [23] proposes an approach where self-managing units coordinate toward a common model, an architectural structure defined using the architectural formalism of Darwin [33]. Each self-organizing component is responsible for managing its own adaptation with respect to the overall system. To do this, each component maintains a copy of the architecture model of the entire system. While this approach provides the advantage of distributed control and eliminates a single point of failure, requiring each component to maintain a global model

and keep the model consistent, which imposes significant performance overhead. Furthermore, the approach prescribes a fixed distributed algorithm for global configuration. We overcome the performance overhead and coordination issue by allowing tailorable global reorganization without imposing a high-performance overhead, but we trade off distributed, localized control of adaptation decision.

4.2 Formal, Dynamic Architectures

A number of approaches focus on modeling and formalizing dynamic systems, rather than mechanisms to enable self-adaption. Our approach builds on formal architectural modeling, using the model within a framework of reusable infrastructures to enable self-adaptation in a target system. Wermelinger and colleagues developed a high-level language, based on CommUnity, to describe architectures, as well as changes over an architectural configuration, such as adding, removing, or substituting components or interconnections [49].

The K-Component model addresses the integrity and safety of dynamic software evolution, modeled as graph transformations of meta-models on architecture [15]. It uses reflective programs called adaptation contracts to build adaptive applications, coordinated via a configuration manager (similar to Le Métayer's approach [31]).

Darwin is an ADL for specifying the architecture of a distributed system, with an operational semantics that captures dynamic structures as the elaboration of components and their bindings in a configuration [33]. Organization of components and connectors may change during execution. The evolving structures of Darwin are modeled using Milner's π-calculus, allowing the correctness of its program elaboration to be analyzed. Together with its π-calculus semantics, Darwin serves as a general-purpose configuration language for specifying distributed systems. ArchWare [37] and PiLar [12] are examples of ADLs that use architectural reflection to model layers of active architectures, allowing separate concerns to be addressed at different layers. These approaches rely on sophisticated reflective technologies to support the active architectures and enable dynamic co-evolution.

These approaches assume that system implementations are generated from the architecture descriptions. In contrast, our approach relies on external mechanisms decoupled from the target system and can therefore be used to add adaptation to existing systems.

4.3 Style-Specific Approaches with Fixed Quality Attributes

A number of architecture-based approaches provide mechanisms to enable self-adaptation (or system reconfiguration) that focus on particular quality attributes of systems, such as performance [6, 27, 32], survivability [50], or that focus on particular architectural styles, for example, [26, 38].

Most closely related to our own work is that of the UCI Research group headed by Taylor [14], and the research of Sztajnberg [47]. As a natural extension of [38], Taylor's group developed an architecture-based runtime architecture evolution framework, which dynamically evolves systems using a monitoring and execution loop controlled by a planning loop. This framework supports self-adaptation for C2-style systems, and evolution of the architecture model uses architectural differencing and merging techniques similar to those used for source code version control. Sztajnberg and Loques developed the CR-RIO framework, which uses a style-neutral ADL (CBabel), architectural contracts to specify execution context, application profiles to describe resource requirements, and middleware to perform architectural reconfigurations based on the specified contracts. CR-RIO demonstrates a formal verification capability but does not appear to support automation of multi-objective adaptations, for example by composing multiple contracts, nor does it address engineering aspects. Our approach can be applied to different classes of systems and can address multiple quality objectives.

Current approaches present a number of limitations and unresolved issues, which are addressed by Rainbow. In particular, where traditional adaptive techniques—for example, the ones based on exception-handling mechanisms and network timeouts—rely only on localized knowledge of system states, we use an architecture-based approach to leverage a more global perspective. While existing approaches do not address the quantity of adaptation and system-level details that engineers grapple with in order to build self-adaptation for their systems, we design a language that encapsulates core self-adaptation concepts and hoists them as first-class building blocks for system engineers to build self-adaptation capabilities. Finally, almost no existing approach provides a systematic, integrated approach to self-adaptation that combines an end-to-end system perspective, style-based adaptation, automation of routine human expertise, and incremental support to developing self-adaptation capabilities; we address this by providing a framework with reusable infrastructures and customizable elements.

5 The Rainbow Approach

Related work provides some of the building blocks for our own research. Software architecture research provides the language, models, and analysis mechanisms to represent and reason about a system's runtime properties; related work in self-healing systems and architecture-based approaches demonstrate the effectiveness of using software architecture for particular classes of systems and fixed quality attributes. What is missing is an approach to self-adaptation that (a) is generally applicable to different classes of systems and quality objectives, (b) allows adaptation to be represented as explicit operational entities and chooses the best one in a principled and analyzable way, and (c) provides an integrated approach that saves engineers time and effort in writing and changing adaptation.

Our approach satisfies the above requirements by (1) providing a framework, called Rainbow, that provides general, supporting mechanisms for self-adaptation,

and which can be tailored to different classes of systems and (2) defining a language, called "Stitch," that plugs into this framework and allows adaptation expertise to be specified and reasoned about, and which can be used to automate and coordinate adaptations to satisfy multiple objectives.

The Rainbow framework is illustrated in Fig. 2. It functions as follows. Monitoring mechanisms—*probes* and *gauges*—observe the running *target system*. Observations are reported to update properties of the architecture model managed by the *Model Manager*. The *Architecture Evaluator* evaluates the model upon update to ensure that the system is operating within an acceptable range, as determined by architectural constraints. If the evaluation determines that the system has a problem, the Evaluator triggers the *Adaptation Manager* to initiate the adaptation process and choose an appropriate repair strategy. The *Strategy Executor* executes the strategy on the running system via system-level *effectors*.

There are three important components to making our solution work: (1) software architecture gives us leverage to make self adaptation general and cost-effective; (2) control theory provides a well understood mechanism for closed-loop system adaptation; and (3) utility theory allows us to pick the most appropriate strategy for repair. Details of each of these are enumerated below.

5.1 The Elements of Rainbow

5.1.1 Software Architecture Model and Style

The first major element of Rainbow is the use of a *stylized* software architecture model to monitor and adapt a target system. Like the blueprint of a building, the

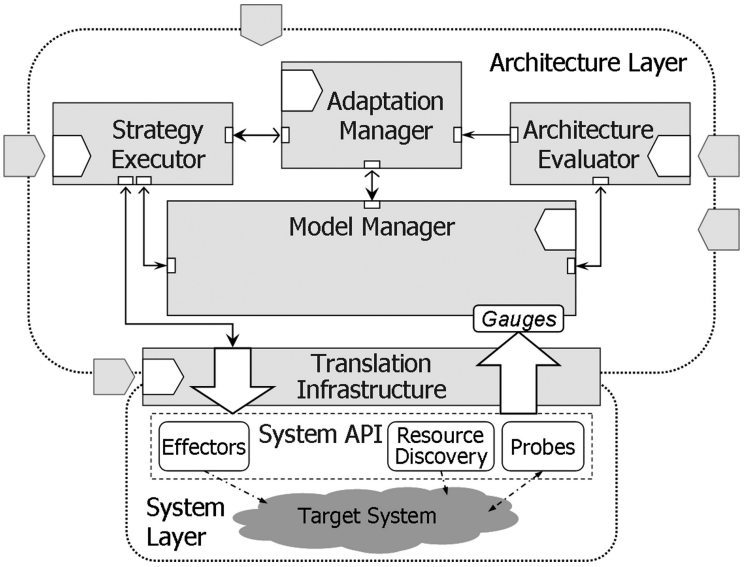

Fig. 2 The *Rainbow framework* with notional customization points

software architecture model of a system provides an abstract view of the modeled software system. The architecture model elides low-level details and allows the architect to focus on the important, high-level properties of the system. The model is described using a particular vocabulary that conveys the structural characteristics of the system, for example, client–server, dataflow, N-tier, and repository. Current approaches to architecture modeling also allow the architect to specify explicit rules, or constraints, about element composition in the system. An architecture model so specified enables the architect to analyze the system for quality attributes such as performance, availability, reliability, and security. Together, *vocabulary*, *rules*, *properties*, and *analyses*, summarized below, comprise the building blocks of *architectural style* [1, 44].

1. **Vocabulary** (V) of element types, including component types (e.g., database, client, server, filter), connector types (e.g., sql, http, rpc, pipe,), and component and connector interface types.
2. **Design rules** (R), or constraints, that determine the permitted composition of those elements. For example, the rules might require every client in a client–server organization to connect to at most one server, prohibit cycles in a particular pipe-filter style, or define a compositional pattern such as a starfish arrangement of a blackboard system or a pipelined decomposition of a compiler.
3. **Properties** (P) that are characteristic of elements in a style, in particular to provide analytic and sometimes behavioral or semantic information. For instance, "load" and "service time" properties might be characteristic of server elements in a performance-specific client–server style, while "transfer-rate" might be a common property in a pipe element of a pipe-filter style.
4. **Analyses** (A) that can be performed on systems built in that style. Examples include performance analysis using queuing theory for a client–server system [46] and schedulability analysis for a style oriented toward real-time processing [3].

While this traditional notion of style suffices to model snapshots of a system's architecture, including dynamic behavior of, and interactions between, system elements (e.g., Darwin [33] and Wright [4]), this characterization of style lacks mechanisms to explicitly represent what dynamic architectural changes are allowed by systems of the style. Capturing allowable operations to the system is important for modeling, analyzing, and reasoning about dynamic system adaptation. For example, knowing whether a system's style allows the activation of a server or the swap of a communication channel helps determine possible adaptations for that system.

To handle the notion of dynamism with respect to architectural structure, we augment the notion of style with *operators*.

5. **Operators** (O). A set of style-specific operations that may be performed on elements of a system to alter its configuration. For example, a service-coalition style might define operators addService or removeService to add or remove a service from a system configuration in this style.

The notion of *architectural style* (augmented with *operators*) gives the architect a powerful abstraction to describe, classify, and analyze many different kinds of systems. Style provides the unifying concepts to factor commonalities out of classes of system and to characterize differences between those classes. Specifically, we leverage style in our design of the Rainbow approach and framework, in combination with the runtime use of architecture and environment models, to achieve generality and cost-effectiveness. We present its design and customization points in Sect. 5.3. Next, we discuss control systems theory, which is integral to the design of our self-adaptation framework.

5.1.2 Control Systems and the Self-Adaptation Cycle

The second major element of Rainbow is the application of control systems concepts to the adaptation problem. Self-adaptation requires a closed loop of control. We choose a specific type of control system model to make our approach generalizable and reusable across diffeent classes of system. In a typical control system, the Controller must have access to relevant *Measured Output* from the target system as well as maintain control over some *Control Input*. In our context, the target system is the software system that requires self-adaptation. Controlling a software system requires mechanisms to obtain information about the system and its execution environment. Therefore, in addition to maintaining a model of the system's architecture, some model of the system's execution environment must also be maintained. Also, the Controller must be able to select a course of action and effect changes on the system.

These required capabilities of control correspond to the $4 + 1$ phases of the adaptation cycle defined by the IBM Autonomic MAPE Architecture mentioned in Sect. 2.1 [17]: *knowledge* is embodied in the architecture model, managed by the Model Manager, *monitoring* is achieved by Probes and Gauges updating the model, *detection* is performed by the Architecture Evaluator assessing problems on the model, *decision* occurs through the Adaptation Manager choosing a remedy based on model states, and *action* is accomplished by the Strategy Executor effecting changes on the system via Effectors. For the *decision* phase, in order to represent and reason about the courses of remedy, we introduce *strategy* as a concept of self-adaptation. Each adaptation decision requires the consideration of multiple factors, which leads to the third element used by Rainbow: utility theory.

5.1.3 Utility Theory

Once a problem is detected by Rainbow, an appropriate adaptation must be chosen. To be effective, such a choice must consider overall business objectives and priorities, and decide between multiple potential adaptations that have possible interacting effects on the system (e.g., an adaptation that fixes performance might affect security concerns, and vice versa). To deal with this, our approach uses utility theory.

To determine the most appropriate strategy in a given circumstance, we need to define values for the objectives, relate the objectives to specific system conditions,

and assess the impact of the strategies on the objectives. One important concern is the uncertainty in the outcome of a particular strategy: enacting a strategy does not necessarily mean that the strategy will be successful on the system. This uncertainty is due to a number of factors, including intervening operation of the system between problem detection and adaptation, inadequate knowledge of the environment, or unanticipated errors in strategy execution. We address this by combining utility theory with a stochastic model of the strategy outcomes. This provides a method to quantify strategies relative to the objectives, under uncertainty.

5.2 Znn.com Example

To illustrate the framework, consider an example news service, *Znn.com*, that serves multimedia news content to its customers, inspired by real sites like *cnn.com* and *rockymountainnews.com*. Architecturally, Znn.com is a web-based client–server system that conforms to an N-tier style. As illustrated in Fig. 3, Znn.com uses a load balancer to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization against service response time. A set of client processes (represented by the *C* component) makes stateless content requests to one of the servers. Let us assume we can monitor the system for information such as server load and the bandwidth of server–client connections. Assume further that we can modify the system, for instance, to add more servers to the pool or to change the quality of the content. We want to add self-adaptation capabilities that will consider monitored information and adapt the system to fulfill Znn.com objectives.

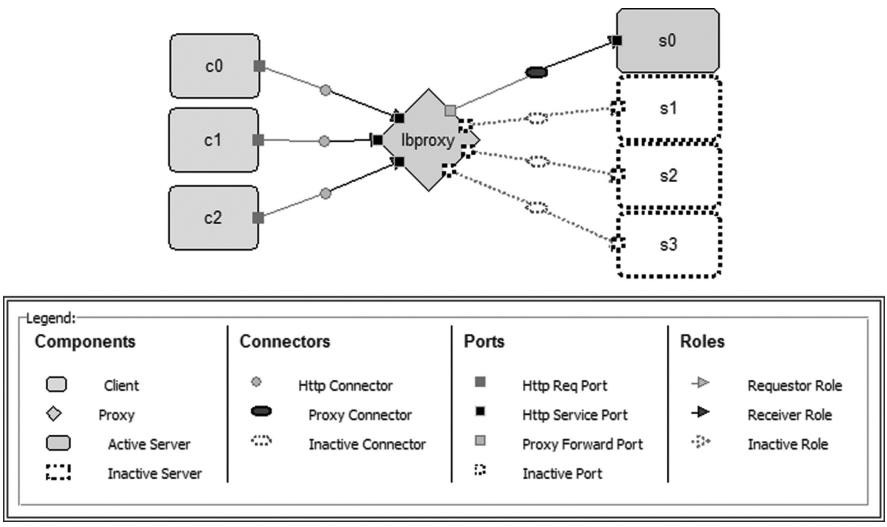

Fig. 3 Architecture model of the Znn.com system

The business objectives at Znn.com require that the system serve news content to its customers within a reasonable response time range while keeping the cost of the server pool within its operating budget. From time to time, due to highly popular events, Znn.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent unacceptable latencies, Znn.com opts to serve only textual content during such peak times in lieu of providing its customers zero service. The Znn.com system administrators (sys-admins) adapt the system using two actions: adjust the server pool size or switch content mode. When the system comes under high load, the sys-admins may increase the server pool size until a cost-determined maximum is reached, at which point the sys-admin switches the servers to serve textual content. If the system load drops, the sys-admin may switch the servers back to multimedia mode to make customers happy, in combination with reducing the pool size to reduce operating cost.

The adaptation decision is determined by observations of overall average response time versus server load. Specifically, four adaptations are possible, and the choice depends both on the conditions of the system and on business objectives:

1. Switch the server content mode from multimedia to textual
2. Switch the server content mode from textual to multimedia
3. Increment the server pool size, and
4. Decrement the server pool size

We want to help Znn.com automate system management to adjust the server pool size or to switch content between multimedia and textual modes. In reality, a news site like *cnn.com* already supports some level of automated adaptation. However, automating decisions that trade off multiple objectives to adapt a system is still unsupported in most systems today. For instance, while automating adaptations on performance concerns is possible (e.g., load balancing), it is much harder to do so in the presence of conflicting qualities such as security.

In terms of Znn.com, the average response time and server load for Znn.com are monitored and those measurements update corresponding properties in the Znn.com architecture model managed by the Znn.com-customized Model Manager. The customized Architecture Evaluator evaluates the model to make sure that no client experiences a request-response latency above a certain threshold. If a client is experiencing above-threshold latencies, the Evaluator triggers the Adaptation Manager to initiate the adaptation process and determine whether to activate more servers or decrease content quality. The customized Strategy Executor carries out the strategy on the Znn.com system using the provided system hooks.

Building a self-adaptive system such as that outlined above is a costly proposition if the important components such as the monitoring, model management, adaptation, and translation mechanisms have to be built from scratch. For this reason, we have engineered an integrated framework with shared infrastructures and developed an iterative process to facilitate reuse of self-adaptive functionalities and reduce the cost and effort of achieving self-adaptation.

5.3 Tailorable Rainbow Framework

Rainbow is a framework with general and reusable infrastructure services that can be tailored to particular system styles and quality objectives, and further customized to specific systems. The customization is notionally illustrated as plug-in pieces in Fig. 2. The *Rainbow framework* consists of a number of components that provide the monitoring, detection, decision, and action capabilities of self-adaptation.

This customizable self-adaptation framework has a number of advantages. Providing a substantial base of reusable infrastructure greatly reduces the cost of development. Providing separate customization mechanisms allows engineers to tailor the framework to different systems with relatively small increments of effort. In particular, the tailorable model management and adaptation mechanisms give engineers the ability to customize adaptation to address different properties and quality concerns, and to add and evolve adaptation capabilities with ease. Furthermore, a modular adaptation language to specify the adaptation policy allows engineers to consider adaptation concerns separately and then compose them.

5.3.1 Rainbow Models

The Rainbow framework leverages two kinds of models to make adaptation decisions: the architecture model and the environment model. An architecture model reflects abstract, runtime states of the target system itself. Many current approaches do not consider the system context, or environment, to make adaptation decisions. Rainbow addresses this shortcoming through an explicit treatment of *environment* states in the self-adaptation process. An environment model provides contextual information about the system, including the executing environment and its resources. For example, if additional servers are needed, the environment model indicates what spare servers are available. When a better connection is required, the environment model has information about available bandwidth on other communication paths.

Managing an executing system dynamically requires knowing the entities that are present, the runtime states they are in, and how they communicate. As noted, the architecture model captures the state of the system as a graph of interacting, communicating entities representing the Component and Connector (C&C) view of architecture [11]. It consists of an instance of the target system defined in a particular style, associated properties and their dynamically updated values, and constraints on the structure of the target system.

The architecture of the system for the Znn.com example is described in the ClientServerFam style with component, connector, and property types for clients, servers, and HTTP connections. Clients in this system define an *average_latency* property value and an architectural constraint specifying that this property should always be below a threshold.

The environment model captures states of the target system's execution environment to provide additional information for the self-adaptation process. Information about the various *resources* must be sufficient to facilitate reasoning about adapta-

tion. As with architecture, we represent environment information as a graph where nodes represent resources and typed edges represent relations between resources, such as physical connection, containment, and dependencies. We capture common relation and resource types in an *environment style*. Environment resources typically relate closely with system elements, so we maintain a mapping between architecture-model elements and environment-model elements.

5.3.2 Translation Infrastructure—Monitoring and Action

In order to get information out of the target system into an abstract model for management, and then to push changes back into the system, the layer marked *Translation Infrastructure* in Fig. 2 provides **monitoring** and **action** (cf., Sect. 5.1.2) hooks, and bridges the abstraction gap between the system and the architecture model. This infrastructure builds on prior work and encompasses monitoring mechanisms, action mechanisms, and various sets of correspondence mappings [5, 10, 22].

Monitoring Mechanisms: *Probes* and *gauges* extract system states, then aggregate and abstract them to update the model. Intuitively, a probe measures some part of the system, while a gauge interprets that measurement to provide a reading. In Rainbow, as illustrated in Fig. 4, probes are deployed onto the target system to measure and publish system information, such as CPU load or process run state. Gauges are associated with specific properties in the architecture model; they collect, aggregate, and abstract probe measurements to populate corresponding architectural properties. Different kinds of probes are deployed onto the target system to detect system states (e.g., whether compression across a communication link is enabled), measure quality attributes (e.g., link latency or intrusion detector state), and discover resources (e.g., to find an available Apache server). Likewise, different types of gauges are needed to aggregate and interpret system properties (e.g., to average latency).

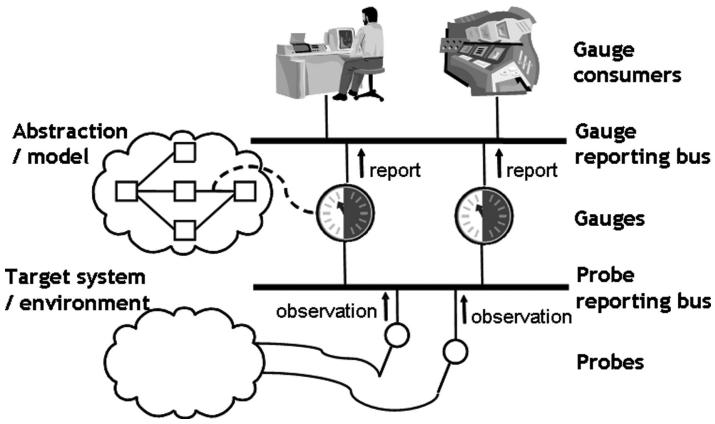

Fig. 4 Monitoring mechanisms: probes and gauges

To tailor the monitoring mechanisms, an adaptation engineer identifies the properties of specific element types to monitor and finds matching gauges and probes from gauge and probe libraries to monitor those properties (or develops them if none are available). The engineer maps the gauge-updated property to the architectural property via the *mapping* attribute, and also defines the target probe, by type name, to which the gauge maps. While we require probes and gauges to enable overall Rainbow functionality, they are not the focus of this chapter.

Action Mechanisms: *Effectors* carry out change operations on the target system; they are associated with architectural operators in the Rainbow Architecture Layer (Fig. 2). Under the hood, the mechanism to realize an effector could range in complexity from a system-call, to a script, to a complex, workflow-based subsystem (e.g., KX Worklets [48]). As with probes and gauges, we require effectors to enable overall Rainbow functionality, but they are not the focus of this chapter.

Rainbow's dependency on monitoring and action capabilities for the target system is not a serious limitation. We build on other researchers' work on probing and effecting capabilities, including adaptive middleware technology [2, 8]. Furthermore, modern systems increasingly support probing and effecting functionalities, as evidenced by products from industry initiatives such as IBM's Autonomic Computing [17] and Microsoft's Dynamic Systems Initiatives [34].

Translation Mappings: Our use of an abstract model to monitor and control the target system requires us to bridge the abstraction gap with correspondence mappings. In a prior publication [10], we identified four distinct kinds of correspondence mappings, maintained by the Translation Infrastructure, to facilitate translation of control information between the architecture model and the target system. For example, when the Strategy Executor invokes an effector, arguments to be passed to the effector must be translated from architectural elements to target-system entities. We briefly summarize the mappings below:

- A **Type** map relates a type of element in the architecture model with a type of entity in the target system, including any properties defined for the type of element/entity.
- An **Element** map relates an element instance in the architecture model with an entity in the target system, including the property values.
- An **Operation** map relates an architectural operator, along with its formal parameters (type and name), to an effector with its corresponding parameters.
- An **Error** map relates the identifier and error sources of an exception in the target system to a corresponding error at the architecture level.

5.3.3 Model Manager

The *Model Manager* manages both the architecture and environment models of the target system. It maintains references between elements of the environment and the architecture models. It tracks the model states, maintains correspondence between the model and the system and environment states via gauges, provides the Rainbow components with shared access to the models via query and modify APIs, and

deploys gauges (and corresponding probes) as dictated by model property queries. Elements in both the architecture and the environment models are accessed via direct model reference in the adaptation scripts (e.g., EnvModel.elementX.prop).

To tailor the Model Manager, it is sufficient to tailor the managed models. A style writer specifies a vocabulary (a family of element types) to describe the architecture of the target system, defines the architecture and environment model instances, and identifies the relevant properties to collect via the monitoring infrastructure.

5.3.4 Architecture Evaluator

Armed with a model that captures runtime system and environment states, we need a mechanism to **detect** when an adaptation is needed (cf., Sect. 5.1.2). When any model property changes, the *Architecture Evaluator* evaluates the conformance of the architecture model to a predefined set of constraints. Upon detecting a constraint violation, it notifies the Adaptation Manager (Fig. 2) to trigger adaptation. This mechanism leverages prior work on the use of architectural constraints, specified in first-order predicate logic, to identify flaws in system design [36]. We extend this work by checking architectural constraints over runtime system properties to detect target system problems at runtime.

To tailor the Evaluator, a style writer specifies as rules the topological and behavioral constraints that (a) characterize the bounds of the target system and/or (b) signify opportunities for adaptation. These architectural rules are specified in the architecture model as first-order predicate logic expressions over architectural structure and properties.

5.3.5 Adaptation Manager

Once a problem is detected, we need a mechanism to **decide** on the appropriate adaptation remedy (cf., Sect. 5.1.2). When triggered by the Architecture Evaluator, the *Adaptation Manager* uses the architecture model to select a remediation strategy that best suits the present problem state of the system, then coordinates the execution of that strategy. Automating system adaptation requires formalizing three kinds of information to instruct the machine to act automatically: for *what* to adapt, *when* to adapt, and *how* to adapt the system.

A quality dimension determines *what* to adapt for and corresponds to a business quality of concern, which is characterized as a utility function and mapped to a monitored architectural property. For example, Average response time (uR) is mapped to ClientT.experRespTime in the architecture and has the utility function defined by the points $\langle (0,1), (500,0.9), (1500,0.5), (4000,0) \rangle$ to represent the utility of average response time at 0, 500, 1500, and 4000 ms. The utility of values of points in between are interpolated. To manage multiple objectives, each quality of concern is given a relative weight that captures business preferences across the quality dimensions. To help decide *when* adaptations are applicable we specify conditions of applicability, e.g., invariant self.avg_latency $<$ MAX_RESPTIME.

```
1   module newssite.strategies.example;
2   import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3   import lib "newssite.tactics.example";
4   import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
5
6   define boolean styleApplies = ...
7   define boolean cViolation = exists c : T.ClientT in M.components |
8       c.experRespTime > M.MAX_RESPTIME;
9
10  strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
11    define boolean hiLatency = ...
12    define boolean hiLoad = ...
13
14    t1: (hiLatency) -> switchToTextualMode() {
15      t1a: (success) -> done ; }
16    t2: (hiLoad) -> enlistServer(1) {
17      t2a: (!hiLoad) -> done ;
18      t2b: (!success) -> do [1] t1 ; }
19    t3: (default) -> fail;
20  }
```

Fig. 5 An example *strategy* SimpleReduceResponseTime

The Stitch self-adaptation language allows strategies to be specified that capture a pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, possibly waiting for the action to take effect. Actions use operators on the architectural style to make changes to the system. A strategy also specifies conditions of applicability that determine in what contexts it should be involved. Furthermore, we need to specify cost–benefit attributes to relate its impact on the quality dimensions. Detailed language features appear in [9].

The adaptation process works as follows: When the *Architecture Evaluator* detects an *adaptation condition*, it triggers the *Adaptation Manager* to initiate a round of adaptation. The Adaptation Manager first checks the strategy *conditions of applicability* to filter a subset of applicable *strategies* based on current system conditions (reflected in the model). In Fig. 5, SimpleReduceResponseTime applies when the conditions styleApplies (definition elided in line 6) and cViolation (defined in lines 7 and 8) are true. The Adaptation Manager then selects the best strategy from the subset by computing the expected utility of each strategy. Briefly, the expected utility of each strategy is computed by first computing the expected aggregate impact of each strategy on each *quality dimension* using the specified *cost–benefit attributes*. Next, the strategies are scored using the *utility preferences* over the quality dimensions. Finally, the highest scoring strategy is selected.

The Adaptation Manager combines utility, decision, and control theories to solve the decision-making problem in self-adaptive systems. To tailor the Adaptation Manager, the engineer specifies a set of adaptation strategies, the quality dimensions and utility preferences, and the cost–benefit attributes to enable automated selection of strategies.

5.3.6 Strategy Executor

Once a strategy is chosen, we need a mechanism that can carry out the adaptation on the target system. The *Strategy Executor* is dispatched by the Adaptation Manager to do this. It resolves model references within the strategy against the Rainbow model, observes model states and evaluates branch conditions to determine operators to execute and corresponding system-level effectors to carry out changes.

The Strategy Executor is tailored by the set of operators of the style. For example, for Znn.com, operators would include addServer, removeServer, and setFidelity.

5.4 Rainbow Application to Znn.com

To illustrate how to customize the Rainbow framework, let us walk through the Znn.com example. Table 1 gives an overview of how each of the Rainbow components is customized for Znn.com. This example is simplified to illustrate only the major features of Rainbow.

The stakeholders in the Znn.com example are the customers and the news service provider. The customers care about quick response time of their news requests and high content quality (i.e., multimedia over textual). While aware of the customer content quality preferences, the provider is constrained by infrastructure provisioning costs to provide the service. We use these three quality concerns to define the quality dimensions, which correspond to measurable properties in the target system. We capture each dimension as a discrete set of values:

1. Response time: low, medium, high
2. Quality: graphical or multimedia
3. Budget: within or over

We elicit from the service providers the utility values and preferences for these dimensions. In addition, since response time is affected by the amount of time required to complete an adaptation, we also need to consider a fourth dimension, disruption, which should be minimized. We use an ordinal scale of 1–5 to express the degree of disruption. Cost–benefit attributes necessary for strategy selection are

Table 1 Znn.com: example application of the Rainbow framework

Set	Rainbow component	Customization content highlight
Objective	Adaptation Manager	timely response (*uR*), high-quality content (*uF*), low-provisioning cost (*uC*)
Vocabulary	Model Mgr, Translators	ClientT, ServerT, DatabaseT, HttpConnT
Property	Architecture Evaluator, Monitoring Mechanisms	ClientT.reqRespLatency, HttpConnT.bandwidth, ServerT.load, ServerT.fidelity, ServerT.cost
Rule	Architecture Evaluator	ClientT.reqRespLatency <= MAX_LATENCY
Operator	Strategy Executor	addServer, removeServer, setFidelity
Strategy	Adaptation Manager	SwitchToTextualMode, SwitchToMultimediaMode, EnlargeServerPool, ShrinkServerPool

Table 2 Znn.com quality dimensions and utility preferences

Label	Description	Architectural property	Utility function	Weight
u_R	Avg Response Time	ClientT.experRespTime	$\langle(\mathsf{low}, 1), (\mathsf{med}, 0.5), (\mathsf{high}, 0)\rangle$	0.4
u_F	Avg Content Quality	ServerT.fidelity	$\langle(\mathsf{textual}, 0), (\mathsf{multimedia}, 1)\rangle$	0.2
u_C	Avg Budget	ServerT.cost	$\langle(\mathsf{within}, 1), (\mathsf{over}, 0)\rangle$	0.3
u_D	Disruption	ServerT.rejectedRequests	$\langle(1, 1), (2, 0.75), (3, 0.5), (4, 0.25), (5, 0)\rangle$	0.1

specified with respect to these four quality dimensions. Given our understanding of the quality dimensions, we can specify discrete utility functions for these four dimensions and complete the utility profiles. To determine the utility preferences, assume that Znn.com considers response time the most important, followed by budget, then content quality, and finally disruption. The quality dimensions and utility preferences are summarized in Table 2.

As part of the N-tier style of Znn.com, a set of element types are defined to model elements of the system architecture: ClientT to model client instances, ServerT for server instances, DatabaseT for databases in the data layer, and HttpConnT as one of the prominent protocols of communication. Properties corresponding to the objectives are defined on the style elements to help measure and assess satisfaction of the objectives; respectively, they are ClientT.reqRespLatency, ServerT.fidelity, ServerT.cost, shown in Table 2. These and other properties are measured by probes and gauges in the translation infrastructure.

A rule specifies the acceptable bound of request-response latencies experienced by a client: exceeding MAX_LATENCY indicates a problem. A set of operators correspond to available effectors in Znn.com: the system can be controlled to add or remove servers, or to change the fidelity of the served content.

When Rainbow is customized as above, during operation the Model Manager deploys gauges and corresponding probes on Znn.com to monitor server status, connection bandwidths, and request-response latencies experienced by the clients (can be approximated via server-side proxy). Probes usually report instantaneous and low-level values, while gauges aggregate and average these measurements and report them as values of corresponding architectural properties to the Model Manager. When the Model Manager updates the architecture model, the Architecture Evaluator checks the model to make sure that the constraint is satisfied, i.e., no client experiences a request-response latency above the maximum threshold.

If a client experiences above-threshold latencies, a constraint violation occurs, and the Evaluator triggers the Adaptation Manager to initiate adaptation. The Adaptation Manager scans through a repertoire of strategies, filtering out the inapplicable ones, then scores them to determine expected utility.

The Znn.com example has four possible strategies, corresponding to each of the adaptations outlined in Sect. 5.2: SwitchToTextualMode, SwitchToMultimediaMode, EnlargeSeverPool, and ShrinkServerPool. We also specify cost–benefit attribute vectors for these strategies, not shown here, that relate the impact of each

Table 3 Znn.com strategies and cost–benefit impact

Strategy	u_R	u_F	u_C	u_D	Utility
SwitchToTextualMode	−2⇒low	−1 ⇒textual	+0⇒within	3	0.75
EnlargeServerPool	−2⇒low	+0⇒multimedia	+1⇒over	1	0.70

strategy to the four quality dimensions. For example, SwitchToTextualMode lowers the response time and the fidelity level, does not affect the cost, and incurs some level of disruption.

Let us assume that Znn.com hits a peak load period, and the system state falls into a problem state in which the response time is high, the infrastructure cost is within budget, and the content mode is multimedia. In this case, only the strategies SwitchToTextualMode and EnlargeSeverPool are applicable. So we need to score the strategies to determine which one to choose given the utility preferences. The cost–benefit attribute vectors would yield aggregate attribute vectors and utility scores for the two strategies as shown in Table 3.

The utility scores indicate DropFidelityStrategy as the better adaptation strategy, given the current system conditions. The Adaptation Manager delegates the execution of this chosen strategy to the Strategy Executor, which evaluates the strategy and invokes the setFidelity operator. This operator is mapped to a corresponding effector to change the Znn.com system. Once changes are effected, Rainbow's adaptation cycle continues to monitor system states.

Note that if Znn.com attributed a lower weight to budget, or a higher weight to disruption, or swapped the importance of disruption versus budget, then the other strategy would have scored higher. Using such utility-based analysis, we can choose a strategy by considering four dimensions and accounting for trade-offs across those using the additional input of business utility preferences.

6 Conclusions and Ongoing Work

In this chapter, we described our approach to architecture-based self-adaptation, which allows engineers to add self-adaptation facilities to existing systems. This approach, called Rainbow, involves adding an external mechanism to monitor and enact changes in systems. We summarized the elements of Rainbow, and how they can be customized to different styles of systems and quality dimensions of interest. Our approach is two-pronged: we provide a framework of reusable infrastructure that can be tailored to particular domains and we provide a language called Stitch that can allow adaptation techniques to be codified. We have given an intuition behind the approach as applied to a simple networked system. Interested readers are referred to [9] for details of the customization and the Stitch language.

As summarized in Table 4, we applied Rainbow to a number of systems, including two small client-server systems (CSSys and UnivSys), a service-coalition system (Libra), and two N-tier systems (Znn.com and the infrastructure of TalkShoe.com). In all cases we applied Rainbow to adapt the target system within some

Table 4 Summary of applying the Rainbow approach

Claim	CSSys	Libra	UnivSys	TalkShoe	Znn.com	SysAdm	Netbwe
General—Rainbow applies to many *styles* and multiple *objectives*?							
− 3+ styles	(CS)	(SvcC)	(CS)	(N-tier)	(N-tier)	(SvcC)	(SvcC)
− 3+ objectives	(*perf*)	(*perf*+*cost*)	(*security*)	(*avail.*)	(4)	(*bw*+*avail.*)	(3)
Cost-effective—Rainbow demonstrates *reuse* (between instances) and *ease of use*?							
− Reusable	√		×	√		×	×
− Easy to use	×	×	×	√ 93h	√ 34h	×	×

×: not applicable, not demonstrated.

user-specified quality goal (e.g., availability), and in the case of Znn.com, we also demonstrated that Rainbow self-adaptation achieved multiple objectives [9]. Finally, we demonstrated that Rainbow could codify system administration tasks.

Our experience with using this approach on a number of systems has pointed to several open areas of future research:

6.1 Improving Detection and Resolution Capabilities

Our approach favors simple but straightforward detection for rapid recognition of problems using a few key variables, for arguably, greater efficiency and effectiveness [25]. Our approach pushes observations into the model and adaptation is triggered when architectural constraints fail. Alternatively, we would like to explore using more sophisticated quality-of-service (QoS) analyzers to continuously evaluate the system for opportunities of improvement based on QoSs.

Our current utility-based approach considers only information about the current state of the system to choose strategies for adaptation. We have implemented mechanisms that consider some simple historical information to avoid repeating bad actions. We would like to take advantage of more historical information and effects, for example, using machine learning as part of the selection process to avoid oscillation and to improve selection quality. We could also integrate learned predictions to anticipate certain QoS problems, such as an anticipated rise in CPU load, drop in available bandwidth, or even a change in the state of user tasks [41].

6.2 Analyzing Adaptation

One natural question that follows from our approach is how to systematically analyze the behavior of the adaptive system and assure certain system properties? Specifically, we would like to develop analyses that answer the following questions:

- Is an adaptation operation consistent with the architectural style? The challenge is to determine the interaction between structure and behavior in an architectural change. This may be addressed, for example, by Kim's work [29].

- We have not addressed the issue of asynchrony in automated system self-adaptation, i.e., the effects of an adaptation takes time to propagate into the system, and the Adaptation Manager must take that delay into account when deciding the next step of adaptation. Can we automatically determine the timing delay of an adaptation operation? The challenge is to formalize *effectors* to enable timing analysis.

6.3 Adapting Adaptation

Currently, the utility preference profile and cost-benefit attributes are statically determined. While they can be changed manually, we would like to be able to change these dynamically as user needs change. To do this, we anticipate integrating formal notions of a user's task [21, 45].

Acknowledgments This material is based up work supported by the US Army Research Office (ARO) under grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's Cylab, and NSF grants IIS0534656 ("Role of Architecture in Facilitating Design Collaboration") and CNS-0615305 ("Activity-Oriented Computing"). Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of these funding agencies.

References

1. Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.
2. ACM. Adaptive middleware. *Communications of the ACM*, 45(6), June 2002.
3. Robert Allen, Steve Vestal, Dennis Cornhill, and Bruce Lewis. Using an architecture description language for quantitative analysis of real-time systems. In *Proc. of the 3rd International Workshop on Software and Performance*, ACM Press, pages 203–210. 2002.
4. Robert J. Allen. *A Formal Approach to Software Architectures*. PhD thesis, Carnegie Mellon University School of Computer Science, May 1997.
5. Robert Balzer. Probe run-time infrastructure. http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt, 2001.
6. Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, volume 3527 of *LNCS*, Springer, pages 1–17, June 13–14, 2005.
7. Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proc. of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, ACM, New York, pages 28–33, 2004.
8. *Proc. of the Working Conf. on Complex and Dynamic Systems Architecture*, December 12–14, 2001.
9. Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. Technical Report CMU-ISR-08-113, Carnegie Mellon University School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213, May 2008.
10. Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. An architecture for coordinating multiple self-management systems. In *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, June 2004.

11. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Views and Beyond*. Pearson Education, Inc., 2003.
12. Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. Dynamic coordination architecture through the use of reflection. In *SAC '01: Proc. of the 2001 ACM Symposium on Applied Computing*, ACM, New York, pages 134–140, 2001.
13. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of WICSA2*, Massachusetts, USA, August 28–31, 2001. Kluwer Academic Publishers, New York.
14. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. [19], pages 21–26.
15. Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *REFLECTION '01: Proc. of the 3rd International Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, Springer-Verlag, London, UK, pages 81–88, 2001.
16. Peter H. Feiler, Bruce Lewis, and Steve Vestal. Improving predictability in embedded real-time systems. Technical Report CMU/SEI-2000-SR-011, Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA 15213, December 2000.
17. A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.
18. David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems*, Lecture Notes in Computer Science, Springer-Verlag, Inc. New York, pages 61–89, 2003.
19. David Garlan, Jeff Kramer, and Alexander Wolf, editors. *Proc. of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, ACM Press, New York, November 18–19, 2002.
20. David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural descriptions of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, Cambridge 2000.
21. David Garlan and Bradley Schmerl. The radar architecture for personal cognitive assistance. *International Journal of Software Engineering and Knowledge Engineering*, 17(2), April 2007. A shorter version of this paper appeared in the 2006 Conference on Software Engineering and Knowledge Engineering (SEKE 2006).
22. David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In CDSA [8].
23. Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organizing software architectures for distributed systems. In Garlan et al. [19], pages 33–38.
24. Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, 2007.
25. Malcolm Gladwell. *Blink: The Power of Thinking Without Thinking*. Penguin, January 2006.
26. Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis. In *Proc. of the 13th International Conf. of Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pages 23–34, May 1991.
27. Michael Hinz, Stefan Pietschmann, Matthias Umbach, and Klaus Meissner. Adaptation and distribution of pipeline-based context-aware web architectures. In *WICSA '07: Proc. of the 6th Working IEEE/IFIP Conf. on Software Architecture*, IEEE Computer Society, Washington, DC, page 15, 2007.
28. IBM. An architectural blueprint for autonomic computing, 2004.
29. Jung Soo Kim and David Garlan. Analyzing architectural styles with Alloy. In *Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA 2006)*, Portland, ME, July 17, 2006.

30. John C. Knight, Dennis Heimbigner, Alexander L. Wolf, Antonio Carzaniga, Jonathan C. Hill, Premkumar Devanbu, and Michael Gertz. The Willow survivability architecture. In *Proc. of the 4th Information Survivability Workshop*, October 2001.
31. Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.
32. Yan Liu and Ian Gorton. Implementing adaptive performance management in server applications. In *Proc. of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, IEEE Computer Society, Washington, DC, page 12, 2007.
33. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM, New York, pages 3–14, 1996.
34. Microsoft Corporation. Dynamic systems initiative. http://www.microsoft.com/\breakwindowsserversystem/dsi/, 2003.
35. Marija Mikik-Rakic, Nikunj Mehta, and Nenad Medvidovic. Architectural style requirements for self-healing systems. In Garlan et al. [19], pages 49–54.
36. Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, 1998.
37. Ronald Morrison, Dharini Balasubramaniam, Flávio Oquendo, Brian Warboys, and R. Mark Greenwood. An active architecture approach to dynamic systems co-evolution. In *ECSA*, volume 4758 of *LNCS*, Springer, New York, pages 2–10. September 24–26, 2007.
38. Peyman Oreizy. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine, 2000.
39. Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptative software. *IEEE Intelligent Systems*, 14(3):54–62, May–June 1999.
40. Robert H. Perry, Don W. Green, and James O. Maloney. *Perry's Chemical Engineers' Handbook*. McGraw-Hill, New York, seventh edition, 1997.
41. Vahe Poladian. *Tailoring Configuration to User's Tasks under Uncertainty*. PhD thesis, Carnegie Mellon University School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213, May 2008.
42. Dale E. Seborg, Thomas F. Edgar, and Duncan A. Mellichamp. *Process Dynamics and Control*. Wiley Series in Chemical Engineering. John Wiley & Sons, New York, 1989.
43. Mary Shaw. Beyond objects: A software design paradigm based on process control. *Software Engineering Notes*, 20(1):27–38, January 1995.
44. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
45. Joao Pedro Sousa. *Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments*. Technical report cmu-cs-05-123, Carnegie Mellon University School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213, March 28, 2005.
46. Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Proc. of the 10th International Conf. on Software Engineering and Knowledge Engineering*, pages 146–151. Knowledge Systems Institute, 1998.
47. Alexandre Sztajnberg and Orlando Loques. Describing and deploying self-adaptive applications. In *Proc. 1st Latin American Autonomic Computing Symposium*, July 14–20, 2006.
48. Giuseppe Valetto, Gail Kaiser, and Gaurav S. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In *8th European Workshop on Software Process Technology*, pages 102–116, June 2001.
49. Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. *SIGSOFT Software Engineering Notes*, 26(5):21–32, 2001.
50. Alexander L. Wolf, Dennis Heimbigner, Antonio Carzaniga, Kenneth M. Anderson, and Nathan Ryan. Achieving survivability of complex and dynamic systems with the Willow framework. In CDSA [8].

http://www.springer.com/978-0-387-89827-8

Autonomic Computing and Networking
Denko, M.; Yang, L.T.; Zhang, Y. (Eds.)
2009, Hardcover
ISBN: 978-0-387-89827-8