Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation

Shang-Wen Cheng

CMU-ISR-08-113

17 May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Garlan, Chair
Jonathan Aldrich
Peter Steenkiste
Jeff Magee, Imperial College London

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, the US Army Research Office (ARO) under grant numbers DAAD19-01-1-0485 and DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab, the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298, and a 2004 IBM Eclipse Innovation Grant.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, the U.S. government, NASA, IBM, or any other entity.

Keywords: Self-adaptive system, software architectural style, adaptation objective, utility preferences, strategy, tactic, architectural operator

To
My almighty God
My dear Family
My beloved Hsien

Abstract

Modern, complex software systems (e-commerce, IT, critical infrastructures, etc.) are increasingly required to continue operation in the face of change, to *self-adapt* to accommodate shifting user priorities, resource variability, changing environments, and component failures. While manual oversight benefits from global problem contexts and flexible policies, human operators are costly and prone to error. Low-level, embedded mechanisms (exceptions, time-outs, etc.) are effective and timely for error recovery, but are local in scope to the point-of-failure, application-specific, and costly to modify when adaptation objectives change. An ideal solution leverages domain expertise, provides an end-to-end system perspective, adapts the target system in a timely manner, and can be engineered cost-effectively.

Architecture-based self-adaptation closes the "loop of control," using external mechanisms and the architecture model of the target system to adapt the system. An architecture model exposes important system properties and constraints, provides end-to-end problem contexts, and allows principled and automated adaptations. Existing architecture-based approaches specialize support for particular classes of systems and fixed sets of quality-of-service concerns; they are costly to develop for new systems and to evolve for new qualities.

To overcome these limitations, we pose this thesis: *We can provide software engineers the ability to add and evolve self-adaptation capabilities cost-effectively, for a wide range of software systems, and for multiple objectives, by defining a self-adaptation framework that factors out common adaptation mechanisms and provides explicit customization points to tailor self-adaptation capabilities for particular classes of systems, for multiple quality-of-service objectives.*

Our approach, embodied in a system called Rainbow, provides an engineering approach and a framework of mechanisms to monitor a target system and its environment, reflect observations into the system's architecture model, detect opportunities for improvements, select a course of action, and effect changes. The framework provides general and reusable infrastructures with well-defined customization points, a set of abstractions, and an adaptation engineering process, focusing engineers on adaptation concerns to systematically customize Rainbow to particular systems. To automate system self-adaptation, Rainbow provides a language, called Stitch, to represent routine human adaptation knowledge using a core set of adaptation concepts.

Acknowledgments

I deeply appreciate my advisor and mentor, David Garlan, for his penetrating insight, patient guidance, candid advice, kind encouragement, inspiring tango, and fine taste of wine. My thanks to Jonathan Aldrich, Peter Steenkiste, and Jeff Magee for carefully reviewing my work to strengthen and elevate its quality, and to Jonathan for going beyond the call of duty to coach me on defining operational semantics. I thank Mary Shaw for her lessons on writing, research, abstraction, & classification.

I am eternally grateful to my dear Mom, Dad, and Sister for loving, supporting, and understanding me; my Gramps, Aunts, and Uncles in Taiwan for shaping and disciplining me during the crucial early years.

And, a man cannot long *and* happily live without his friends.
My cheers!
My heartfelt thanks, to...

George Fairbanks, a fantastic housemate
for enjoying beers, watching movies, sharing good times,
giving advice, exchanging ideas, helping me to crystallize the news site example

Bradley Schmerl and Karen Hay
for mentorship and friendship, for asking the tough questions not only in research,
but also in romance, over coffee, with sushi and other nourishments.

Carl and *SoYoung Paradis*, siblings-in-Christ
for food, fun, (spiritual) fellowship, and friendship.

Nicholas Sherman
for brews, cools, tools, and drools shared of all instances and types.

Kevin B, *Nels B*, *Greg H*, *Ciera J*, *Annie L*, *Dean S*, and *fellow PhD'ers*
for goodwill, rapport, stimulating interations, and beers.

Vahe Poladian
for helping me to clarify the language formalism based on utility theory;
for Piling Higher & Deeper™, motivating, predicting, steaking, pair-writing.

Jichuan Chang, *Joao Sousa*, *Bridget Spitznagel*, and members of the ABLE group;
An-Cheng Huang and *Ningning Hu*
for brainstorming, collaboration, feedback, idea exchange, and support.

Paul H, *James H*, *Ross M*, *Frank L*, *Kevin L*, *Matt F*, *Roger C*, *David H*, *Ping C*
for the memorable corporate living at D-1, cooking, praying, tan-painting Kevin...

Sharon Pan, my beloved
for her companionship timely, her encouragements gentle, her care loving.

For lending me an ear or a hand;
For showing me who I am;
I appreciate my fellow friends and mentors, whom I cannot all name.

Contents

1 Coping with Change **1**
- 1.1 Self-Adaptation Loop of Control . . . 2
  - 1.1.1 External, Feedback Control . . . 3
  - 1.1.2 IBM Autonomic Framework . . . 5
- 1.2 Architecture-Based Self-Adaptation . . . 6
- 1.3 Opportunities for Improving the State-of-the-Art . . . 7
  - 1.3.1 Lack in System Context and Adaptation Knowledge . . . 8
  - 1.3.2 Lack of Support for Quality-of-Service Trade-Off . . . 8
  - 1.3.3 High Cost of Development and Maintenance . . . 9
- 1.4 This Thesis . . . 10
  - 1.4.1 Thesis Evaluation Plan . . . 11
  - 1.4.2 Thesis Contributions . . . 12
  - 1.4.3 Document Roadmap . . . 12

2 Related Work **13**
- 2.1 Software Engineering and Architecture . . . 13
- 2.2 External Contributing Disciplines . . . 15
- 2.3 Related Self-Adaptation Approaches . . . 17
  - 2.3.1 Adaptive Technologies . . . 17
  - 2.3.2 Industrial Initiatives and Autonomic Computing . . . 19
  - 2.3.3 Architecture-Based Adaptation . . . 19
- 2.4 Limitations to State-of-the-Art Addressed . . . 22

3 Rainbow Overview **23**
- 3.1 Overview of Approach . . . 23
  - 3.1.1 Software Architecture Model and Style . . . 24
  - 3.1.2 Control Systems and the Self-Adaptation Cycle . . . 25
  - 3.1.3 Utility Theory . . . 26
  - 3.1.4 Design Constraints for Self-Adaptation . . . 27
- 3.2 Znn.com Example . . . 27
- 3.3 Tailorable Rainbow Framework . . . 29
  - 3.3.1 Rainbow Models . . . 30
  - 3.3.2 Translation Infrastructure—Monitoring and Action . . . 32
  - 3.3.3 Model Manager . . . 34

3.3.4 Architecture Evaluator . . . 34
3.3.5 Adaptation Manager . . . 35
3.3.6 Strategy Executor . . . 35
3.4 Rainbow Application to Znn.com . . . 36
3.5 Adaptation Engineering Process . . . 37
3.6 Summary . . . 38

4 Stitch Self-Adaptation Language 39
4.1 Rainbow Context for Language . . . 39
4.2 Requirements for the Self-Adaptation Language . . . 40
4.2.1 Nature of System Administration Tasks . . . 40
4.2.2 Language Design Considerations . . . 42
4.3 Self-Adaptation Concepts of Stitch . . . 43
4.3.1 Overview . . . 43
4.3.2 Quality Dimensions, Utility Preferences, and Adaptation Conditions . . . 45
4.3.3 Operator . . . 46
4.3.4 Tactic . . . 48
4.3.5 Strategy . . . 50
4.3.6 Strategy Selection . . . 52
4.4 Semantics of Stitch Constructs . . . 53
4.4.1 Model of Adaptation . . . 53
4.4.2 Utility-Based Strategy Selection . . . 56
4.4.3 *Adaptation Execution* . . . 58
4.5 Stitch Illustration Using Znn.com . . . 66
4.6 Summary . . . 68

5 Customizable Framework 69
5.1 Architecture and Design of Rainbow . . . 70
5.1.1 Rainbow Deployment Architecture . . . 71
5.1.2 Model Manager and Rainbow Models . . . 74
5.1.3 Translation Infrastructure—Monitoring and Action . . . 76
5.1.4 Architecture Evaluator . . . 80
5.1.5 Adaptation Manager . . . 81
5.1.6 Strategy Executor . . . 83
5.2 Adaptation Integrated Development Environment . . . 84
5.2.1 Stitch Script Editor . . . 85
5.2.2 Rainbow Control Console . . . 85

6 Examples and Supporting Evidence 87
6.1 Basic Client-Server System . . . 87
6.2 Libra Videoconferencing System . . . 90
6.3 University Grade System—Security Domain . . . 93
6.4 TalkShoe . . . 97
6.4.1 Background . . . 98

6.4.2 TalkShoe Infrastructure . . . 98
6.4.3 Problem Scenarios for Adaptation . . . 99
6.4.4 Data and Result . . . 103
6.4.5 Conversations with the TalkShoe Architect . . . 104
6.4.6 TalkShoe Summary . . . 105
6.5 Znn.com News System . . . 106
6.5.1 Motivation: *Slashdot Effect* . . . 106
6.5.2 Rainbow Customization for Znn.com . . . 107
6.5.3 Experimental Setup . . . 109
6.5.4 *Slashdot Effect* Traffic Profile . . . 110
6.5.5 Data and Results . . . 111
6.5.6 Znn.com Summary . . . 114
6.6 Interview with System Administrators . . . 114
6.6.1 Methodology . . . 114
6.6.2 Interview Results . . . 115
6.6.3 Adaptation Analysis from Almossawi's Administrative Experiences . . . 119
6.6.4 Interview Summary . . . 120
6.7 Real-World Adaptive Scripts in Stitch . . . 120
6.8 Summary . . . 126

7 Thesis Evaluation **127**
7.1 Claim: Generality . . . 128
7.2 Claim: Cost-Effectiveness . . . 129
7.3 Claim: Transparency . . . 133
7.4 Summary . . . 136

8 Discussion of Issues and Limitations **137**
8.1 Central Control . . . 137
8.2 Asynchronous Interaction and Uncertainty . . . 141
8.3 Closed-Loop Feedback Control . . . 143
8.4 Stitch Expressiveness: Operator, Tactic, and Strategy . . . 144
8.5 Limitations to Adaptation Using a Model . . . 146
8.6 Limitations to Using Utility Theory . . . 147
8.7 Framework, Reuse, and Experience on Cost-Effectiveness . . . 148
8.8 Summary . . . 148

9 Conclusion and Future Work **149**
9.1 Thesis Contributions . . . 149
9.2 Future Work . . . 150
9.2.1 Short-Term Framework Improvements . . . 150
9.2.2 Medium-Term Rainbow Research Issues . . . 151
9.2.3 Longer-Term Research Beyond Rainbow . . . 155
9.3 Summary . . . 156

Bibliography **159**

A Rainbow Framework Architectural Style **173**

B Stitch Grammar **175**

C Znn.com Customization Content **177**

D Additional Thesis Supporting Materials **191**
D.1 Personal Records 191
D.2 Listing and Abstract Descriptions of *netbwe* Subroutines 192

List of Figures

1.1 A closed-loop control paradigm . . . 3
1.2 Conventional block diagram showing a Feedback Control System . . . 4
1.3 The IBM Autonomic MAPE Reference Model . . . 5

3.1 Architecture model of the Znn.com system . . . 28
3.2 The *Rainbow framework* with notional customization points . . . 29
3.3 Example definition of the architecture model in Acme . . . 30
3.4 Example definition of the environment model in Acme . . . 31
3.5 Monitoring mechanisms: probes and gauges . . . 32

4.1 Stitch grammar highlights (see Appendix B on page 175 for full grammar) . . . 47
4.2 An example *tactic* switchToTextualMode . . . 49
4.3 An example *strategy* SimpleReduceResponseTime . . . 51
4.4 Flowchart of the Rainbow adaptation process; Figure 4.5 refines *Execute strategy* 64
4.5 Flowchart of strategy execution, Figure 4.6 refines *Execute tactic* . . . 65
4.6 Flowchart of tactic execution . . . 65

5.1 Rainbow architectural diagram . . . 69
5.2 Rainbow run time deployment . . . 72
5.3 Architectural decomposition of the Model Manager . . . 74
5.4 Visual connection of gauges to the architecture model . . . 76
5.5 Architectural decomposition of the Architecture Evaluator . . . 80
5.6 Architectural decomposition of the Adaptation Manager . . . 81
5.7 Mock-up of a Rainbow AIDE workbench . . . 84

6.1 A class of web-based client-server systems . . . 88
6.2 Architecture model of the client-server system . . . 89
6.3 Experiment result, with and without Rainbow adaptation . . . 91
6.4 Architecture model of the Libra videoconferencing system . . . 92
6.5 Architecture model of the University Grade System . . . 94
6.6 Excerpted Acme description of the University Grade System . . . 95
6.7 Example adaptation strategies for the university grade system . . . 97
6.8 A simplified architecture model of TalkShoe's infrastructure . . . 99
6.9 Tactic notifyByEmail and Strategy NotifyOnExpire . . . 103
6.10 Architecture model of the Znn.com system . . . 108

6.11 Graph of actual, peak-day traffic of a site experiencing *Slashdot effect*. 110
6.12 Graph summarizing preliminary Znn.com experiment data 112
6.13 A system context diagram of the *netbwe* subsystem 120
6.14 Model of the CMU network infrastructure 122

8.1 Control graphs for representative points 138
8.2 GFS Architecture (extracted from [GGL03]) 139
8.3 Control graph for Rainbow 140

9.1 Rainbow illustrated as a feedback control system 156

List of Tables

2.1 Characterization of the space of control paradigm 16

3.1 An example gauge instance specification 33
3.2 Summary of four Translation Infrastructure correspondence mappings 34
3.3 Znn.com: example application of the Rainbow framework 37

4.1 Stitch self-adaptation concepts motivated by the sys-admin's tasks 43
4.2 Data schema for a Utility Profile 45
4.7 Znn.com utility profiles and preferences 66
4.8 Znn.com tactic cost-benefit attribute vectors 67
4.9 Znn.com aggregate attribute vectors for two applicable strategies 67
4.10 Znn.com utility evaluation for two applicable strategies 67
4.11 Traceability summary of Stitch language features 68

5.1 Rainbow architectural style description — Main Family 70
5.2 Who-Does-What-How summary of framework customization 71

6.1 TalkShoe scenario feasibility assessment 100
6.2 TalkShoe scenario development activity data 104
6.4 Summary of Walter's solution strategy and tactics 116

7.1 Task-based estimation of effort: Rainbow versus custom-solution 133
7.2 Summary of example evidence toward thesis evaluation 136

8.1 Invocation relationship between Stitch operational constructs 145

Chapter 1

Coping with Change

Imagine a world where a software engineer could take an existing software system and specify objectives, 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 latency. The engineer 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 keeping down cost of operating the infrastructure. Still another engineer might make a cluster of servers self-adaptive with respect to certain security attacks. Imagine that the engineers could achieve their tasks within a few days to weeks, rather than weeks to months. Imagine further that the engineers could share and reuse the adaptation expertise and quickly apply others' adaptation strategies to their own system.

Increasingly, systems must have the requirement to self-adapt with minimal human oversight. They must cope with variable resources, system errors, and changing user priorities, while maintaining as best as they can the goals and properties envisioned by the engineers and expected from the users. 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 highly specific to the application and tightly bound to the code. As a result, self-adaptation in today's systems is costly to build, often taking many man-months to develop or retrofit systems with the capabilities. Moreover, once added, the capabilities are difficult to modify and usually provide only localized treatment of system errors.

The vision outlined above requires an approach that makes it possible for engineers to easily define adaptation policies that are more global in nature, and which 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 needing to rewrite 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 dissertation, we describe an approach to achieving the above goals using architecture-based self-adaptation techniques. In particular, our approach reflects observed properties of an executing system into properties of an architecture model, where they can be reasoned about using an array of existing architectural analysis techniques. The results of these analyses can then be used to reason about changes that should be made to the target system to correct it or improve its achievement of quality attributes, using utility theory to make trade-offs across these attributes.

Our approach is embodied in a system called Rainbow, which focuses on two means of achieving cost-effective self-adaptation: (1) an approach and mechanism that helps reduce engineering effort and (2) an explicit representation of adaptation knowledge. Rainbow provides an engineering approach and a framework of mechanisms to monitor a target system and its executing environment, reflect observations into its architecture model, detect opportunities for improvements, select a course of action, and effect changes. By leveraging the notion of *architectural style* to exploit commonality between systems, the framework provides general and reusable infrastructures with well-defined customization points to cater to a wide range of systems. It also provides a useful set of abstractions to focus engineers on adaptation concerns, facilitating the systematic customization of Rainbow to particular systems.

To automate system adaptation in general, focusing on mundane and routine system administration tasks in particular, Rainbow provides a language, called Stitch, to represent routine human adaptation knowledge using a core set of adaptation concepts. It offers modularity with respect to quality dimensions and domain expertise, allows specifying multi-step strategies of tactics with conditions of applicability and expected effects, provides a mechanism to tailor adaptation policies to particular system domains, and uses utility theory to determine the best adaptation strategy in the face of uncertainty.

In the following chapters, we make a case for architecture-based self-adapting systems, survey the research landscape, introduce the Rainbow approach and the Stitch language, describe how the approach addresses current limitations and achieves the stated goals, discuss the research and engineering challenges, and demonstrate the approach with example applications focusing on adaptations to improve qualities such as fidelity, performance, security, and cost of operations.

1.1 Self-Adaptation Loop of Control

In the past, systems that required self-adaptation were rare, confined mostly to domains like telecommunication switches or deep space control software, where shutdown for maintenance or upgrade was not an option and manual intervention was not always possible. Today, more and more systems have this requirement and are supporting some form of self-adaptation. Systems such as those in the e-commerce and mobile embedded system domains must operate continuously with only minimal human oversight. They must cope with *variable resources* (e.g., bandwidth and service availability), *system faults* (e.g., server component failing or connections going down), and *changing user priorities* (e.g., high-fidelity video streams at one moment and rapid response time at another). Ubiquitous computing, where highly mobile users operate within heterogeneous environments contending for constrained resources, further motivates the need for self-adaptive systems. However, where self-adaptation capabilities are provided, they are often

built-in, solution-oriented, and application-specific. Such "low-level" adaptation mechanisms—program exceptions, network time-outs, etc.—are great for detecting problems quickly, but they lack an end-to-end system context. Moreover, the adaptation logic is often dispersed throughout the implementation, making modification and maintenance of adaptation functionality costly, analysis of its outcome challenging, and its reuse nearly impossible.

In the Information Technology (IT) domain, self-adaptation support has gained ground but has been limited to custom-made or vendor-specific solutions (e.g., Microsoft Operations Manager [Mic08]). These tools often target a particular aspect of system management, such as reporting and scripting, requiring significant human involvement to monitor and maintain the overall system. While humans are better at understanding the overall problem context than computers, human operators are prone to long reaction time, errors, and varying and possibly inconsistent expertise. The lack of solid automation support for mundane and routine tasks contribute to high IT operation cost. In fact, industry data indicate that the cost of ownership of IT systems attributable to managing the system ranges from 70–90 cents per dollar [Fry03, GC03, Sco02]. Recognizing these challenges, leading software companies like IBM [GC03] are pursuing ways to develop "self-managing and self-provisioning" infrastructure to help businesses streamline IT operations [Fry03].

1.1.1 External, Feedback Control

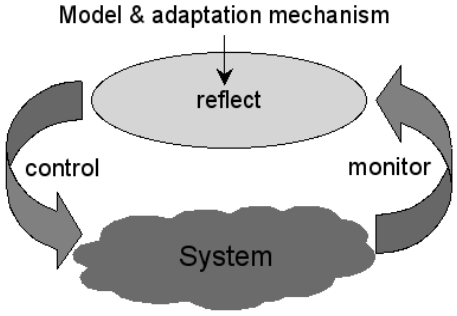


Figure 1.1: A closed-loop control paradigm

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 attributes remain relatively unchanged. In most cases, if something goes wrong, i.e., the system operates beyond acceptable bounds, humans can intervene, often by restarting the failed subsystem, or in the worst case, taking the system offline for repair. However, to allow a system to self-adapt dynamically, a number of researchers have proposed an alternative approach that uses external mechanisms to maintain a form of closed-loop control over the target system (e.g., [OGT+99, GR91, KHW+01]). As shown notionally in Figure 1.1, 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 [SEM89].

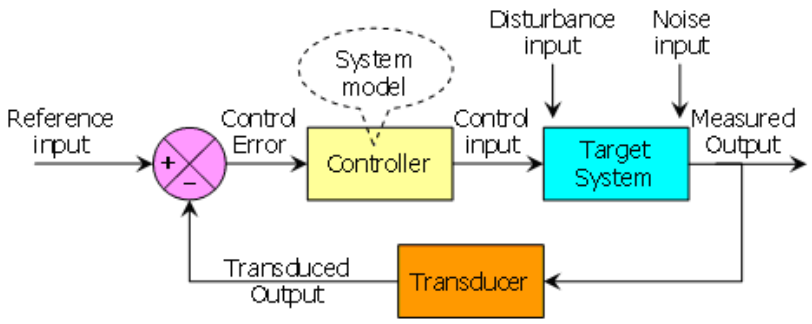


Figure 1.2: Conventional block diagram showing a Feedback Control System

Figure 1.2 shows the block diagram of a single-input, single-output feedback control system. Given the desired setting, called the *reference input*, the Controller modifies the *control input* to the Target System to align the system's *measured output* with the *reference input*. The output is optionally filtered by a Transducer to smooth or aggregate signals.

To illustrate how a feedback control system works, consider an ordinary thermostat-controlled, home-heating furnace. The furnace is the system under control. The temperature is the measured output of the home (system) environment. The thermostat—the controller or adaptation mechanism—controls the *on* and *off* states of the furnace, which comprise the *control input* (or knob) of the system. Via the thermostat, the homeowner can set the desired temperature (say, 68 degrees) as the reference input. The physical model of temperature and thermodynamics form the *system model*. Typical of such systems, the model is considerably simpler than the system being monitored—i.e., the temperature model is much simpler than the furnace subsystem.

The thermostat measures the temperature of the environment and checks it against the set point. If the measured temperature falls below 68, the thermostat turns the furnace on until the measured temperature reaches 68, at which point the thermostat turns the furnace off. For simple systems like these, the control model may be built-in to the design. In more complex systems, such as a chemical processing plant, an explicit process model is necessary for effective control [SEM89]. For example, an air conditioning system for a large building that monitors multiple locations and controls multiple heating or cooling units would require an explicit model of the building partitions and temperatures to efficiently control which units to turn on and when.

For software systems, the external controller requires an explicit model of the target system to reflect on observations and to configure and repair the system [OGT+99]. 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. 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 that of adaptation (or "exceptional") behaviors. With the adaptation mechanism as a separate entity, engineers can modify and extend it, and reason about its adaptation logic, with ease. Furthermore, the separation of mechanisms allows the application of this technique even to legacy systems with inaccessible source code, assuming that the target system provides, or can be instrumented to provide, hooks to ex-

tract 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.

A prominent, commercial computing system that uses this external, feedback control paradigm is the Google File System (GFS). GFS is a massively distributed, highly scalable system that exemplifies a remarkable, custom-built self-adaptive system whose centralized, master controller possesses global knowledge of abstract system states. It runs on commodity server hardware, which has high failure rate. Yet, it serves up data quickly and reliably. It uses a lightweight master controller, combined with server health states, to determine how to serve data from and store data to appropriate disks [BDH03, GGL03]. We aim to generalize a software engineering solution that is characteristically similar, yet applicable to a broad spectrum of system domains.

1.1.2 IBM Autonomic Framework

The reference standard from the IBM Autonomic Computing Initiative codifies an external, feedback control approach in its Autonomic Monitor-Analyze-Plan-Execute (MAPE) Model [IBM04]. Figure 1.3 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 phases of self-adaptation:

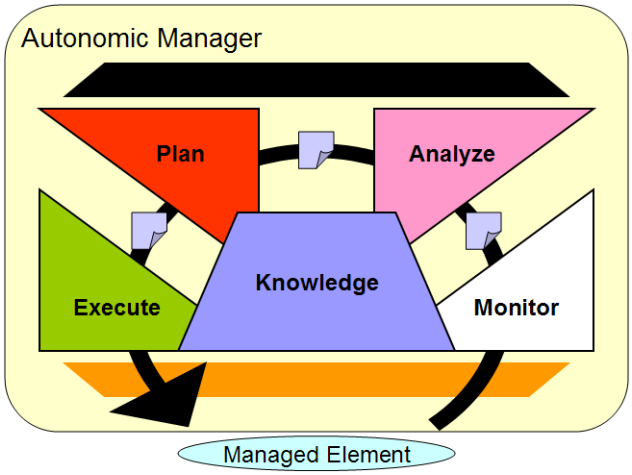


Figure 1.3: The IBM Autonomic MAPE Reference Model

1. **Monitor**: The monitoring phase is concerned primarily with extracting information—properties or states—out of the managed element. Mechanisms range from source-code instrumentation to non-intrusive communication interception.
2. **Analyze**: The analysis phase is concerned with determining if something has gone awry in the system, usually because a system property exhibited a value outside of expected bounds, or a property exhibited a degrading trend. This thesis terms it the **Detection** phase.

3. **Plan**: The planning phase is concerned with determining a course of action to adapt the managed element once a problem is detected. This thesis refers to it as the **Decision** phase.
4. **Execute**: The execution phase is concerned with carrying out a chosen course of action and effecting the changes in the system. In this thesis we refer to it as the **Action** phase.

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. As discussed in Chapter 3, the Rainbow framework provides components that fulfill each of these four phases and the *knowledge* to support self-adaptation.

1.2 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. State machines, queuing theory, graph theory, differential equations, and other mathematical models [SEM89, PGM97] have all been used for model-based, external adaptation. 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 for figuring out what remedial actions to take to fix the system. As we highlight here and discuss further in Section 3.1.1, software architecture strikes an ideal balance as a model for self-adaptation.

Over the past decade and a half, software architecture has emerged as a prominent and powerful design abstraction [BCK98]. A recent branch of work advances the use of the target system's architecture as the external model for dynamic adaptation [GKW02]. The architecture of a software system is an abstract representation of the system as a composition of computational elements and their interconnections [SG96]. Specifically, an *architecture model* represents the system architecture 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 [GMW00] and xADL [DHT01].

The use of software architecture as the basis of a control model for self-adaptation, termed *architecture-based self-adaptation* in this thesis, 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.

Consider, for example, a signal-processing system implemented as a graph of stream-processing elements. The architecture model of the system can be represented as a pipe-filter model where the stream-processing elements are filters and the data-flow connections are pipes.

[1] Although there are different views of architecture [CBB+03], in this thesis we are primarily interested in the component-connector view because it characterizes the abstract state and behavior of the system at run time to enable reasoning about problems and courses of adaptation.

This model of the runtime system configuration provides a global perspective on all the elements of the system. It exposes such important properties as the throughput of each filter and the bandwidth of each pipe, allowing one, for instance, to compute the overall throughput of the system. Furthermore, the model might be associated with explicit constraints on the architecture that, for example, forbid cycles. This knowledge can be used at run time to reason about the effect of a change on the system's throughput or structure.

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) [AAG95]. 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 use real-time schedulability analysis [FLV00].

Style is important because it constrains and gives structure to the system design space. Each style provides opportunities for specific analysis of system behavior and properties. 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. Consider again the signal-processing system. The pipe-filter 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 different 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 an important 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 a client between different servers to reduce latency.

1.3 Opportunities for Improving the State-of-the-Art

A number of researchers have explored self-adaptation using an external architecture model with varying success. Gorlick and colleagues have developed the Weaves framework, which supports continuous observation and dynamic rearrangement of systems in the data-flow architectural style [GR91]. Magee and colleagues have designed the Darwin formalism for specifying software architectures that support dynamic component initiation and binding [MDEK95]. Taylor and colleagues have developed an architecture evolution framework to enable runtime system modifications that are kept consistent with an architecture model, based primarily on hierarchical publish-subscribe in C2 [TMA+96, OMT98].

These architecture-based self-adaptive systems have been hand-crafted to support a particular class of system (e.g., data-flow) and address a specific domain of concern (e.g., performance). Given a system in a supported style, there is typically an ADL and tooling support to model and analyze the system and capture constraints on system properties, and mechanisms to detect

constraint violations and adapt the system. While these efforts represent point solutions in the general problem space of architecture-based self-adaptation, they also highlight three classes of limitations with the state-of-the-art: (1) a lack in system context and adaptation knowledge, (2) lack of support for quality-of-service trade-off, and (3) high cost of development and maintenance. In this section, we consider each class of limitations in turn.

1.3.1 Lack in System Context and Adaptation Knowledge

Since the onset of dynamic systems research in the late-90s, advances have been made in basic monitoring and action technologies crucial for self-adaptation [CDS01, GKW02]. Currently we can dynamically obtain system information and change system states. From machine learning and decision theory communities we have techniques to analyze what changes to make as well as reason about what course of action to take [Mit97, Bat00]. However, we are missing two crucial self-adaptation capabilities—an explicit model for the system context or execution environment, and a systematic and explicit representation for adaptation knowledge—with the following consequences:

1. The lack of an explicit model for the system's environment creates "blind-spots" in adaptation decision-making. In particular, choosing an adaptation requires knowledge of the availability of alternative components, spare resources, etc. Knowledge of options that are not available (a more challenging problem) is likewise important to determine what course of actions are out of question. Finally, knowing about contextual computation elements and resources could help identify opportunities for improvement.
2. The lack of a systematic and explicit representation for adaptation knowledge makes it difficult to capitalize on system management expertise already available today. Furthermore, this reflects the lack of explicit abstractions for reasoning about and controlling self-adaptation capabilities.

To address these limitations, we need a way to model system environment that captures spare resources and alternative components. We also need a representation that hoists adaptation concerns as *first-class*, manipulable concepts to give software engineers control over the self-adaptation capabilities of their system and to allow reasoning about self-adaptation decisions.

1.3.2 Lack of Support for Quality-of-Service Trade-Off

Due to monetary, time, and various resource constraints, keeping a system operational often requires a system administrator to balance multiple, possibly conflicting objectives. Ultimately, a self-adaptation system must also be able to balance multiple quality dimensions simultaneously (e.g., performance vs. cost vs. security). To that end, a custom-solution self-adaptive system might attempt to reconcile all the quality concerns (or *objectives*) at once, such as keeping a system secure while minimizing impact on performance and maintaining availability.

Systems today typically wire in trade-offs on qualities of service, making these *policies* implicit artifacts of design. In fact, a process such as the Architecture Trade-off Analysis Method (ATAM) explicitly considers qualities of service during software design, but does not tradition-

ally expose the resulting trade-off decisions as first-class (operational) entities in the functionality of the software. Hardwiring trade-off policies suffers from a number disadvantages:

1. Adding a new quality of concern may be difficult, requiring changes that touch many aspects of the system, including those that detect and signal changes in the system properties related to the particular quality and those that compute trade-off and decide an outcome.
2. Hardwired policies have a single context of use. For instance, a policy may have embedded a particular *preference* of system response time over quality of delivered content, which becomes obsolete when that preference changes due to evolving business needs.
3. The implicit nature of hardwired policies makes it difficult to reason about satisfying objectives, for there is often (a) no explicit treatment of objectives, (b) no traceability from objectives to the implemented trade-offs, (c) no link between trade-off computation steps (if the computation itself is obvious) and particular objectives, and (d) no obvious distinction between desired versus emergent trade-off outcome.

For a self-adapting system, trade-off policies must be made explicit to enable dynamic adaptation. Different kinds of adaptations will have different quality-of-service trade-offs. For example, an adaptation that addresses reliability and performance might be able to achieve both with redundant, replicated servers. Another adaptation that addresses performance and security might need to make a choice between reducing network response time and increasing the number of intrusion detection filters. Yet another adaptation that addresses cost, performance, and security might need to give different preferences to each depending on the business context. A self-adaptation approach should be flexible enough to cater to different trade-off needs.

In short, current approaches address only fixed quality dimensions, lack support for composition and trade-off across multiple dimensions, and do not cater to varying business contexts. To address these limitations, we need a systematic approach, along with corresponding mechanisms and techniques, to (a) allow developing each adaptation separately for separate quality concerns, (b) facilitate reasoning about and *composing* two or more adaptations that trade off across multiple dimensions to produce the desired objectives, and (c) enable flexible modification of adaptation policies to cater to evolving business context.

1.3.3 High Cost of Development and Maintenance

Building an architecture-based self-adaptive system is a costly proposition because one has to develop the probing infrastructure to monitor the target system, a representation to encode the architecture model and constraints, a manager for the architecture model instance(s), a problem detector to determine adaptation opportunities, adaptation mechanisms to resolve problems and propagate changes to the system, and translation mechanisms to bridge the system-model abstraction gap. Finally, one has to integrate all of these parts into a coherent, self-adaptive system.

Most existing approaches have associated description languages, adaptation mechanisms, and toolsets targeting a single style and fixed quality concerns. Applying these approaches to a new system, often with a different style or different concerns, would require either re-developing the adaptation mechanisms from scratch, or reusing some parts and custom-building the remaining capabilities. Both incur significant time and effort to develop, integrate, and engineer the adaptation mechanisms correctly. Once built, modifying or enhancing the adaptation mechanisms

would likewise incur significant effort.

In short, whether developing a software system ground-up with adaptation support, or otherwise retrofitting adaptation capabilities into an existing system, both are guaranteed to incur high cost. To make economic sense, we need a generalized self-adaptation framework that amortizes cost and effort across multiple systems (which has been the aim of projects such as AEM [DHT02] and IBM autonomic systems [IBM04]), that can cater to more than one style of systems and different quality concerns, that enables engineers to avoid re-developing significant mechanisms, that allows low-effort incremental development of adaptation capabilities.

1.4 This Thesis

Ideally, we would like a solution that realizes the benefits of architecture-based self-adaptation highlighted in Section 1.2 while overcoming the limitations outlined above. The solution should support models and mechanisms that provide a global system perspective to adapt the system effectively and in a timely manner. The solution should leverage domain expertise, allow the specification and evolution of adaptation policies for different domains of concern, and support the composition of policies across domains. Furthermore, the solution should enable system engineers to engineer self-adaptation for their systems cost-effectively.

In this dissertation we develop a new approach to engineering architecture-based self-adaptive systems that allows engineers to tailor a common infrastructure to particular system domains and quality objectives. Specifically, we investigate the following thesis:

> **We can provide software engineers the ability to add and evolve self-adaptation capabilities cost-effectively for a wide range of software systems and for multiple objectives by defining a self-adaptation framework that factors out common adaptation mechanisms and provides explicit customization points to tailor self-adaptation capabilities for particular classes of systems and quality objectives.**

This thesis statement leads to three specific claims that must be demonstrated. First, our approach to architecture-based self-adaptation applies to a wide range of architectural *styles* of systems for multiple objectives or *dimensions* of system quality that architects are typically concerned with. We will refer to this claim as the **generality** of the approach.

Second, compared with custom, style-specific solutions, a framework with common, reusable infrastructures significantly reduces the *cost* of engineering self-adaptation capabilities for existing systems, while explicit customization points makes the framework easy to tailor to target systems. We will refer to this claim as **cost-effectiveness**.

Third, the ability to tailor self-adaptation capabilities implies **transparency** in the knowledge and process of self-adaptation: (a) separating concerns of adaptation makes the distinct concepts and steps of the process *understandable*, (b) explicit adaptation concepts makes actions separately definable and *composable* to achieve overall system goals, and (c) providing a value system makes adaptation choice *automatable*. We summarize the 3 claims as follows:

- **Generality**: The approach applies to a broad spectrum of *styles* for common quality *dimensions* with which modern architects are concerned.

- **Cost-effectiveness**: The approach significantly reduces *cost*, relative to existing specialized solutions, to engineer and evolve self-adaptive systems.
- **Transparency**: The approach makes the adaptation process *understandable*, actions *composable*, and adaptation choice *automatable* for routine system adaptations.

To demonstrate this thesis, we will focus on a particularly important subgroup of computing systems, those currently requiring human administration to ensure normal and uninterrupted operation. Because of the high cost of system upkeep (as much as 70–90 cents per dollar of ownership in some domains), automating self-adaptation for managed systems would reduce administrative cost for IT and e-commerce systems and significantly reduce ownership cost.

This thesis comprises a two-part solution: (1) a framework, Rainbow, to enable architecture-based self-adaptation and (2) a language, Stitch, to express system-administration adaptation concepts. The framework consists of a common, reusable infrastructure with systematically defined components for system monitoring, model management, constraint evaluation, strategy selection, and adaptation execution. The solution approach defines explicit customization points as well as recommends a workflow to tailor the framework for specific systems. The language represents important adaptation concepts as manipulable and *first-class* operational entities. In concert with the framework, the language allows engineers to focus on adaptation concerns when specifying adaptation strategies and policies.

1.4.1 Thesis Evaluation Plan

To evaluate this thesis, we apply the approach and framework to a number of systems, each belonging to one of three architectural styles and addressing up to three quality dimensions of interest. Together, these example applications serve to demonstrate applicability across a breadth of styles as well as transparency. We then assess the cost-effectiveness of the framework using a task-based estimation of effort and informal user feedback. Specifically, we satisfy the three claims as follows.

To evaluate **generality**, we demonstrate that the Rainbow framework applies to multiple styles, for multiple quality dimensions. For practical reasons, we target only a subset of typical styles and quality dimensions. Specifically, we show an in-depth demonstration of the Rainbow adaptation cycle on one system, and provide example applications showing how customization pieces of the Rainbow framework are specified for representative styles of systems.

To evaluate **cost-effectiveness**, we demonstrate *reuse* and *ease-of-use* with the Rainbow framework. We show that the Rainbow framework provides common and reusable infrastructures, which are flexible to customize to make a system self-adaptive. In effect, Rainbow saves engineers time and development effort to add and evolve self-adaptation capabilities to a target system. To show effort savings, we characterize the self-adaptation tasks and provide coarse-grained, task-based estimation of effort, then qualitatively assess and evaluate savings of self-adaptation effort with Rainbow relative to current practice.

To evaluate **transparency**, we demonstrate that the approach allows domain experts to define adaptation strategies separately for different quality dimensions, and enables an *adaptation engineer* to compose strategies across the dimensions and automate adaptation choice. In particular, we show that the approach facilitates understanding how to engineer self-adaptation capabil-

ities. We show that the self-adaptation language provides constructs to specify strategies for different dimensions, support to combine those dimensions meaningfully, and mechanisms to automatically select and carry out adaptations that integrate the strategies to achieve multiple objectives. We demonstrate the expressiveness of the language by (a) interviewing system administrators to understand the administrative process and (b) qualitatively analyzing how well the self-adaptation language represents typical classes of system administrative tasks.

1.4.2 Thesis Contributions

This thesis advances the state-of-the-art in software engineering by improving our understanding of the approaches, mechanisms, and tradeoffs for software architecture-based self-adaptation. In brief, contributions of this dissertation include:

- characterization of a general *approach* to architecture-based self-adaptation customizable to a class of systems and a set of quality dimensions, *language* to represent system administration concepts and adaptation knowledge, and *process* for adaptation engineering;
- demonstration of *techniques* for self-adaptation, including a utility-theoretic algorithm for selecting adaptation strategies, a Markov Decision Process framework to model the adaptation process, and control theory for adaptation control;
- provision of a packaged *tool* set for the Rainbow approach, in particular a mechanism for adaptation selection and interpreter for the Stitch language;
- demonstration of coverage of a representative region of the self-adaptation system space with particular styles of system, domains of concern, and kinds of feasible adaptation.

1.4.3 Document Roadmap

In the remainder of this document, Chapter 2 describes the background of this work and discusses related solutions and their limitations. Chapter 3 restates the thesis requirements, describes the overall architecture-based self-adaptation approach, and sets the context for introducing the self-adaptation concepts and language in Chapter 4. Chapter 5 presents the customization points of the framework and illustrates how they facilitate reuse and low-effort customization of self-adaptation for a target system. Chapter 6 details example applications and evidence in support of the thesis, which Chapter 7 evaluates. Then, Chapter 8 discusses issues and limitations, and Chapter 9 concludes this thesis, highlighting future work.

Chapter 2

Related Work

In this chapter, we describe work related to the self-adaptation approach in this thesis. We start with the contributing disciplines of software architecture and analysis, control theory, and decision theory. We then discuss related approaches for architecture-based self-adaptation.

2.1 Software Engineering and Architecture

In the software engineering discipline, a large body of foundational work on software architecture paved the way for architecture to be used as a model to reason about a software system. The landmark paper by Perry and Wolf defined software architecture and established it as a discipline, drawing analogies from building architectural styles and forming a basis for using architectures as system models [PW92]. Shaw and Garlan characterized and codified many common styles of system architecture [SG96]. Bass, Clements, and Kazman investigated the practical issues of applying software architecture through many case studies, providing techniques for designing and analyzing architecture [BCK98, BCK03]. Here we briefly discuss a directly relevant subset of the discipline—architecture style, architecture description languages, and quality attributes.

An *architectural style* defines, for a class of systems, a vocabulary of element types, properties common to the element types in theses systems, a set of constraints on the permitted composition, and the associated analyses for reasoning about this class of systems [AAG93, MKMG97]. Style has been formalized and applied in many system designs. In essence, architectural style is useful for capturing commonalities between systems of a particular class and, consequently, variabilities across different classes of systems.

A body of work on analytical models from the areas of engineering and computer sciences have contributed to analysis for software, usually associated with particular styles. Examples of style-specific system analyses include queuing theory and performance modeling for client-server systems [LGSZ84, SG98], throughput analysis for data flow networks [SC81], and reliability models [XDP04]. In this work, we use a software architecture model to leverage its power for abstracting system structures and for expressing and analyzing system properties.

A central theme to the work of the ABLE Group[1] is to tailor a generic design environment

[1] Architecture-Based Languages and Environments Group, led by Professor David Garlan in the School of Computer Science, Carnegie Mellon University

to specific classes or domains of system through the explicit use of architectural styles. Acme, a generic ADL, supports the explicit notion of style [GMW00] and provides a first-order predicate logic similar to UML OCL [Obj01] to capture system design constraints [Mon99]. The Acme constraint logic provides a set of architectural functions to facilitate the definition of logical expressions that capture such relationships as connectedness, type conformance, and hierarchy. AcmeStudio [Sch08, SG04] is a customizable design environment and visualization tool for software architectural designs based on Acme [CMU08].

This thesis takes the notion of architectural style into the runtime world. To support the needs of runtime adaptation, as presented in Section 3.1.1, we augment the notion of style and the associated tools with the notions of operators to change an architecture. Many ADLs have been developed for various domains, modeling purposes, and even interchange, including Acme, C2, MetaH, Rapide, SADL, UniCon, Wright, and XML variants like xAcme [CU01] and xADL [DHT01]. A few efforts have attempted to compare and classify them [Cle96, MT97]. Acme is used in this thesis because of its explicit support for style, particularly architectural constraints, and its features for extensibility.

Wright provides an explicit capability to define styles in a fairly formal fashion [All97]. C2 and associated tool suites support the description and analyses of event-based, hierarchical publish-subscribe systems with explicit rules of inter-hierarchy communication [MORT96]. Darwin models systems with bidirectional communication links and supports formal reasoning about deadlocks and concurrency [MDEK95]. Weaves uses a data-flow style that facilitates software construction and analysis, allowing parts of systems to be snipped and spliced without disruptions to data-flow [GR91]. SADL describes systems as interfaces and connections, and supports direct translation to logic sentences for correctness analysis [MR97]. There are also XML variants of ADLs that are meant to support interchange as well as style-generic architecture descriptions, including xArch [UC01], xAcme [CU01], ADML, and xADL [DHT01].

A number of efforts have attempted to codify architectural design expertise. Butler developed an approach and model for assessing security risks and quantifying such risks as relative index [But02]. Bass and John characterized software usability issues as architectural design trade-offs [BJ03]. Project ArchE used software analytical models to assist the software architect in making trade-offs during system design [BBKS05]. Bass, et al., described specific architectural design tactics to achieve system quality attributes [BCK03].

A *quality attribute* is an attribute of the system that is orthogonal to its functionality and often bears on and pervades the structure of the system [BLKW97]. Examples of quality attributes include performance, security, scalability, and ease of maintenance—the "ilities." Quality attributes play a critical role in the design of software architecture, and design *tactics* can be used to achieve the desired quality attributes. A tactic is a "design decision that influences the control of a quality attribute response." A collection of tactics forms an *architectural strategy*, and tactics can be packaged into *architectural patterns*.

This thesis applies the design-time concepts of quality attributes and tactics to support adaptation decisions in the runtime space, using *adaptation tactics* and *adaptation strategies* to achieve the desired system quality attributes at run time. We primarily address concerns that directly impact the runtime qualities of a system, including performance, security, dependability; we do not address such design-time concerns as modifiability, maintainability, and extensibility.

2.2 External Contributing Disciplines

Here we highlight four areas of research—control theory, decision theory, system and network management, and artificial intelligence—from which we have incorporated concepts and techniques to develop our approach to self-adaptation.

Control Theory Closed-loop control is one of several process control paradigms extensively studied and applied in various disciplines to influence the behavior of *dynamical systems*[2]. Closed-loop control typically involves a controllable process or "target system" (e.g., a heat exchanger water tank) that usually includes the environment, a controlled variable or "measured output" (e.g., the outflow of a heat exchanger), and a load or "control input" (e.g., the inflow to a heat exchanger) [SEM89, Wik08a]. The objective of closed-loop control is to continuously adjust the control input into the target (e.g., amount of inflow), or the process itself (e.g., amount of heat), or both, so that the measured output matches the reference input (e.g., outflow temperature, T) to within some error margin.

Monitoring the measured output to determine what changes to make is known as *feedback* control, while monitoring only the control input to determine control action is known as *feedforward* (or predictive) control. Feedforward control can anticipate changes to the measured output, while feedback control reacts to perturbations in the target, so the techniques are often combined. A specific form of closed-loop control called *model predictive control (MPC)* is particularly well suited for multi-input, multi-output (MIMO) control problems, where significant interactions occur between manipulated inputs and outputs [PGM97].

Controlling a software system can be viewed as a MIMO control problem; however, the measured output and control input are often discrete rather than continuous as in traditional closed-loop process control. Fortunately, control theory also allows the control of discrete variables. Closed-loop control thus seems well-suited overall for the control of software systems, as applied in this research, provided that the measured output and control input can be properly identified. Other interesting forms of control not pursued in this research include open-loop control, self-stabilization, multi-level supervisory control, and biological homeostasis. Open-loop control assumes reliable conditions and does not monitor the process. Self-stabilizing systems are designed to converge to a desired behavior despite arbitrary starting state [Dol00]. Advances in adaptive robotics demonstrate the power of multi-level supervisory control to gradually enable autonomy [BMD+03]. Homeostatic systems constantly move away from the "region" of bad behavior toward the healthy region [Sha02].

This thesis draws concepts and techniques—stability, accuracy, settling time, overshoot—from the control-theory domain to control software systems. Unlike traditional control systems, however, this thesis targets discrete, state-based computing systems [HDPT04], which require a different type of system model, such as an architecture model.

There are two fundamental aspects to any control system: the availability of information and the degree of control over the target system. Let us consider these as two orthogonal dimensions—scope of knowledge and scope of control. The spectrum for *scope of knowledge*

[2] A *dynamical system* is a mathematical formalization for any fixed "rule" which describes the time dependence of a point's position in its ambient space [Wik08c].

ranges from localized to global, while that of *scope of control* ranges from singleton (conceptually, one *knob* of control) to multi-point. By examining the extremes, we can characterize the space grossly as four representative points, as shown in Table 2.1.

Table 2.1: Characterization of the space of control paradigm

Scope	**Local Knowledge**	**Global Knowledge**
Singleton Control	Micro-reboot	reboot
Multi-point Control	S-O-S	GFS, Rainbow, ...

The most restrictive point in this space is a controller with global knowledge of the system, but only a single knob of control. A crude example is the common user experience with Microsoft Windows, where the user has global knowledge of whether the operating system has failed (the blue screen of death), but the user only has a single control action: to reboot. Consider a more elaborate system where local knowledge of system components is available, but only a single control action is possible. For instance, in a J2EE application server, an operator could obtain fine-grained knowledge about bean health and micro-reboot beans that fail, thereby reducing disruption to the rest of the system and increasing overall system availability [CKF+04].

Some systems provide more degrees of control than just reboot, including capabilities to swap components and tune quality of service. Depending on the scale and configuration of such systems, a controller might have global knowledge about the states of all of the system parts, or only local knowledge to states of subsystems. Rainbow and the Google File System serve as examples of the former kind. An example of a system whose controller has only local knowledge is the Self-Organizing System proposed by Georgiadis, Magee, and Kramer, where distributed component managers coordinate and construct local models of the system and manage local components individually [GMK02]. Another example is the Aura framework with an Environment Manager component that possess knowledge and access to local contexts and resource configurations [PGS+07, SG02].

While useful for comparison, these four points represent only a rough dissection of a large space, and there may be many other interesting points in between. Also, no single point has absolute advantage over the others. For example, in contrast to a system with multi-point control and global knowledge, a system with singleton control and local knowledge boasts greater simplicity in mechanism. Furthermore, a system with multi-point control and local knowledge is potentially more scalable and less susceptible to a single point-of-failure than a system with multi-point control and global knowledge.

Decision Theory and Utility Decision theory is an area of study concerned with how real or ideal decision-makers make (or should make) decisions, and how optimal decisions can be reached [Bat00, Wik08b]. A core concern of decision theory is to study choice (between incommensurable commodities) under uncertainty. Numerous techniques for choice under uncertainty have been developed, including the isomorphic forms of decision tree and decision table [Bak04],

the use of utility [Wik08e] (or expected value) to attribute values to decisions, and Markov Decision Process to solve for an optimal decision [How60].

In this thesis, we extend the notion of utility to compare quality dimensions, and we adopt preference weights to assign relative importance to the dimensions. Together, utility and preferences enable our automating the choice of adaptation strategy under uncertainty.

System and Network Management A major research challenge in system and network management is to allow dynamically adaptable policies that cater to different user needs in distributed, cross-domain systems [SL02]. Ponder is one notable example of a policy language for specifying both system management and security policies with dynamic adaptability of behavior [DDLS01]. In particular, Ponder assumes distributed, object-oriented systems, and provides an object-based enforcement mechanism as well as a deployment model [DLSD01]. Our work shares a similar goal of supporting "dynamic adaptability of behaviour by changing policy without recoding or stopping the system." However, our approach extends beyond system management to provide a general engineering framework for self-adaptive systems.

Artificial Intelligence The area of artificial intelligence contributes a number of potential techniques to enable system self-adaptation, including rule-based expert systems and the use of fuzzy logic [Dur94], planning algorithms [RN03] (e.g., FLECS [VS95]), and reinforcement learning [Mit97] (e.g., Q-learning [Wat89]). In our approach, rather than automatically generating an optimal adaptation plan, we use an explicit language of representation to capture routine human expertise about system adaptation. Our self-adaptation language embodies rule-like constructs (condition-action), but relies on utility theory to choose an adaptation, instead of an inference engine to "reason" about the most appropriate action, given the current target system state.

2.3 Related Self-Adaptation Approaches

This section highlights related work in self-adaptive systems. We start with adaptive technologies, explore initiatives from industry and IBM Autonomic Computing in particular, and examine similar architecture-based approaches to system self-adaptation.

2.3.1 Adaptive Technologies

The self-adaptation approach advanced in this thesis builds on a number of adaptive technologies, as described below. Specifically, some of the research address the sensing and effecting problems, which are not the main focus of this thesis.

As part of the DASADA project, researchers developed a number of adaptive technologies and defined standards for the probe and gauge infrastructures [Bal01, CDS01, GSC01]. To monitor software states, Heineman developed the *AIDE* probe architecture [Hei98] to instrument Java source code and Wells and Pazandak developed *ProbeMeister* [PW02, WP01] to instrument Java bytecode. Researchers at Columbia University developed *gaugents* to collect and propagate monitoring events and advanced a *Worklets* effector technology to effect changes using mobile Java code [GGK+01, VKK01, VK02]. Event systems like SIENA [CRW01] and MEET [Gro02]

provided the communication infrastructure for event distillation and packaging—supporting such features as defining event models or event time windows, and detecting event protocol anomaly—to enable monitoring of self-healing system in a distributed setting. Workflow systems have been applied to support planning-based self-adaptation, such as the Cougaar-based adaptive system of BBN Technologies [CV02].

Some self-adaptation research provided enabling adaptive techniques as well as primitives for reasoning about adaptations, from generic, low-level architecture operators to stylized operators [MRMM02, MM03]. Advances in research to bridge architecture to code (e.g., ArchJava [ACN02] and transformational connector [Spi05]) contributed techniques for discovering and constructing architectural models from system events (e.g., DiscoTect [YGS+04]).

The mechanism for doing actual adaptations embody different concepts and vary greatly, from doing only quiescent adaptation, to supporting mixed-mode adaptation where old and new components are allowed to communicate [BK07], to micro-rebooting components of a managed system periodically to improve overall system availability [CKF+04] (also known as *recovery-oriented computing*). Finally, the separation of policy from mechanism is a common adaptive technique, also applied in designing the approach in this thesis. An example from a related work is to abstract service-specific knowledge as *recipes*, separate from the lower-level adaptation mechanism, to allow cost savings with a shared framework that synthesizes the optimal service configuration [Hua04].

While this thesis advances a form of self-adaptation external to the controlled software system and notionally "above the application layer," there is a similar body of research in which adaptation is applied from "below the application layer" at the infrastructure or operating system level. In particular, adaptive components or multi-fidelity components provide useful capabilities in existing software systems and offer complementary approaches to self-adaptation [SN01, FPS02, JBB03, MLR03]. In addition, a recent branch of middleware research attempts to support dynamically adaptive distributed systems by developing reflective, adaptive, and, in general, more "intelligent" middleware [Agh02, KCBC02]. Adaptive middleware technology may prove synergistic with the approach proposed in this thesis.

Adaptive middlewares monitor and control software applications using interception or interposition techniques. Specifically, an adaptive middleware makes extensive use of interceptors to, for example, profile, trace, and even affect dynamic library usage [Cur94, NMMS99]. Fault-tolerant CORBA provides transparent OMG-compliant fault tolerance through strong replica consistency, using techniques such as N-versioning, hot, warm, or cold swap, and redundant servers [NMMS02]. Some of the challenges include the ordering of operations, duplication of operations, recovery, and consistency in the face of multithreading. Some adaptive middlewares explicitly support and adapt for quality-of-service (e.g., Quality Objects (QuO) [LSZ+01]) or tracks changes in user, system, and environment contexts (e.g., MADAM [MPF+06]).

From monitoring and effecting technologies to adaptive middlewares, this collection of research advances specific niches of adaptation capabilities to enable the overall adaptation engineering framework explored in this thesis. Moreover, not addressed by any of the above effort, this thesis addresses issues of engineering and of making the self-adaptation process transparent to the target system owners and developers.

2.3.2 Industrial Initiatives and Autonomic Computing

Several industry initiatives aim to achieve self-managing or autonomous systems to increase system autonomy and reduce the growing business overhead of IT operations, including the N1 Architecture by Sun Microsystems [Sun02], the PlantCare project by Intel Research for the autonomous care of houseplants [Int02], the Adaptive Enterprise by Hewlett-Packard [Hew03], the Dynamic Systems Initiative by Microsoft [Mic03], and the Utility Data Center jointly by HP and Microsoft [Tur03]. Most notable is IBM's ten-year Autonomic Computing initiative from 2002 [GC03, JBB03, MLR03].

The primary challenge of Autonomic Computing is to control self-managing components in a large, distributed system to produce emergent autonomic behavior. Autonomic computing applies the concept of a control loop to monitor, evaluate, and adapt a system component. At the heart of the control loop are various kinds of business and system knowledge to help determine how to adapt the behavior of the system. Researchers of autonomic computing distinguish levels of control, differentiated primarily by degree of automation and scope of knowledge. Self-management is categorized into *self-configuring*—to adapt automatically to dynamically changing—*self-healing*—to discover, diagnose, and react to disruptions—*self-optimizing*—to monitor and tune resources automatically—and *self-protecting*—to anticipate, detect, identify, and protect from any attacks. A suite of tools has emerged from IBM's effort, most notably the Autonomic Computing Toolkit [IBM08], which provides consoles and tools to help problem diagnosis and engineer autonomic systems.

This thesis applies a similar control loop paradigm to monitor and adapt software systems. A principle difference of this approach is the use of an explicit architecture model combined with adaptation strategies and utility-based adaptation choice to achieve similar autonomic capabilities. Similar to the business goals of IBM and others, this thesis aims to support cost-effective engineering of self-adaptation capabilities. In particular, this thesis solves a number of difficult issues: the representation of adaptation expertise, a framework to facilitate cost-effective engineering, the automated choice of adaptation that achieves multiple objectives, and the development of a cost-effective approach to apply self-adaptation.

2.3.3 Architecture-Based Adaptation

To date, several dynamic software architectures and architecture-based adaptation frameworks have been proposed and developed [BCDW04, GSRU07, OGT+99], including an effort to characterize the style requirements of *self-healing systems* [MRMM02]. Below, we examine a representative set of approaches, categorizing each by its primary focus, then highlighting its main features. Broadly speaking, related approaches either focus on formalism and modeling, or mechanisms of adaptation. A third category uniquely addresses distribution and decentralization of control.

Distributed, Decentralized Adaptation Work on self-organizing systems [GMK02] proposes an approach where self-managing units coordinate toward a common model, an architectural structure defined using the architectural formalism of Darwin [MK96]. Each self-organizing component is responsible for managing its own adaptation with respect to the overall system and

requires the global architecture model to do so. 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 imposes significant performance overhead. Furthermore, the approach prescribes a fixed distributed algorithm for global configuration. This thesis overcomes 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.

Formal, Dynamic Architectures A number of approaches focuses on modeling and formalizing dynamic systems, using graph rewriting or a flavor of π-calculus, showing minimal mechanisms, if any, for either enabling the modeled dynamism in a target system or enforcing certain adaptation properties, such as integrity and safety, on a target system. While this thesis does not formalize dynamic systems, our approach builds on a formal architectural model, 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 (sometimes described as an Architecture Modification Language), based on CommUnity, to describe architectures and for operating changes over an architectural configuration, such as adding, removing, or substituting components or interconnections [WLF01]. The language follows an imperative style and builds on a semantic domain where architectures are modeled through categorical diagrams, and dynamic reconfiguration through algebraic graph rewriting. The language provides two kinds of commands: basic commands perform the actual configuration, while composite commands only control the flow of execution.

The K-Component model addresses integrity & safety of dynamic software evolution, modeled as graph transformations of meta-models on architecture [DC01]. It uses reflective programs called *adaptation contracts* to build adaptive applications, coordinated via a configuration manager (similar to Le Métayer's approach [Le 98]).

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 [MK96]. Organization of components and connectors may change in the architecture of the system during execution. The evolving structures of Darwin is elegantly modeled using Milner's calculus of mobile processes, allowing the correctness of its program elaboration to be analyzed. Together with the π-calculus semantics, Darwin serves as a general-purpose configuration language for specifying distributed systems.

ArchWare [MBO$^+$07] and PiLar [CdlFBS01] are example of ADLs that use architectural reflection to model layers of active architectures, allowing separate concerns to be addressed at different layers. The 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 in this thesis relies on external mechanisms decoupled from the target system and can therefore be used to add adaptation to existing systems.

Style-Specific Approaches with Fixed Quality Attributes A number of architecture-based approaches provided frameworks of mechanisms to enable self-adaptation (or system reconfig-

uration). We highlight a few below. While they vary in details, the approaches share a number of common characteristics: they generally apply a closed-loop control and use an architecture model for reasoning about the target system; and they assume certain structures in the target system and adapt for a fixed set of quality attributes.

For example, Gorlick and Razouk developed the Weaves framework, which supports continuous observation and dynamic rearrangement of systems in the data-flow style to facilitate software construction and analysis, allowing parts of systems to be snipped and spliced without disruptions to data-flow [GR91]. Wolf, et al., developed the Willow framework to enhance the *survivability* of critical, networked information systems, consisting of network sensors to acquire network state, synthesis and diagnostics components to analyze the sensor events and respond with network changes, an environment to coordinate a workflow of network changes, and a universal actuation interface [WHC+01].

Peyman Oreizy's dissertation on the "open architecture software" approach proposes the use of an application's architectural model as a basis for decentralized software evolution for a greater degree of adaptability while supporting increased consistency over prior techniques [Ore00]. In his approach, an *architecture evolution manager* validates changes to the architecture model, in the C2 hierarchical publish-subscribe style, and carries out the changes on the application's implementation to reflect the model. An associated ArchStudio environment provides a number of tools to support evolution of software via changes to the architecture model for C2-style applications. As a natural extension, the UCI research group then developed an architecture-based runtime *architecture evolution framework*, which dynamically evolves systems using a monitoring and execution loop controlled by a planning loop [DHT02]. 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.

Bastista and colleagues at Lancaster developed the Plastik adaptation framework, which uses an extension of the Acme ADL to specify adaptation policy, relies on OpenCOM APIs to sensing system state and actuating changes, and focuses on performance properties [BJC05]. Hinz, et al., demonstrated an adaptive framework for pipelined information systems that adapts for throughput and other performance goals [HPUM07]. Liu and Gorton developed the Adaptive Server Framework, which applies IBM's autonomic framework for constructing dynamic and adaptive application servers that optimize performance [LG07].

Mukhija and Glinz developed the CASA framework to adapt for resource availability concerns in mobile network environments [MG04]. Each application offers alternative component configurations. The generic concept of *contract* is used to define adaptation policy for each application. Interactions between applications are mediated by a *service negotiator*, which participates in a service agreement protocol (SAP) to determine the application's configuration. The adaptation and enforcement system of the framework handle adaptations in response to changes in the execution environment, and some of the typical adaptations react to either abundance or scarcity of resources.

Sztajnberg and Loques developed the CR-RIO framework, which uses a style-neutral ADL (CBabel), architectural *contracts* to specify execution context, application *profile* to describe resource requirements, and a middleware to perform architectural reconfigurations based on the specified contracts [SL06]. CR-RIO demonstrates formal verification capability but does not appear to support automation of multi-objective adaptations, e.g., by composing multiple contracts,

nor to address engineering aspect.

This thesis generalizes the application of the closed-loop control by providing an engineering framework that can be tailored to different styles to enable adaptation for multiple quality dimensions.

2.4 Limitations to State-of-the-Art Addressed

In this chapter, we presented an overview of the related work to show how the state-of-the-art partially serves our thesis objective. Advances in software research provide the language, model, and analysis to represent and reason about a system's software architecture, giving us the powerful notion of architectural style. Advances in adaptive technology provide mechanisms to enable self-adaptation, in particular to monitor system states and effect changes, which form building blocks for advanced system control. Research in related self-healing systems and architecture-based approaches demonstrate point solutions for particular classes of systems and singular quality dimensions.

However, current approaches present a number of limitations and unresolved issues, which we address in this thesis. In particular, where traditional adaptive techniques—e.g., ones based on exception-handling mechanisms and network time-outs—rely only on localized knowledge of system states, we use an architecture-based approach to leverage 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 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 an engineering framework with reusable infrastructures and customizable elements.

The insight of using *architecture-based feedback control* to develop self-adaptive systems still leaves some important questions to address. What kind of control model does one develop and how best to use it? What are the most efficient and effective ways to get information out of the target system, and how does one interpret the system states? How does one make decisions of and reason about the remedial actions to take on the system? How can one best effect changes on the system? There is also the overall challenge of engineering such a system cost-effectively.

In this thesis, we focus on two core challenges to achieve cost-effective self-adaptation:

1. **Approach and mechanism** to reduce engineering effort
2. **Representation of adaptation knowledge** critical to the decision process for choosing remedial actions

We now present our overall self-adaptation approach in Chapter 3.

Chapter 3

Rainbow Overview

The objective of this thesis is to empower software engineers with an approach and the tools to add self-adaptation capabilities to software systems. In Chapter 2 we argued how the state-of-the-art partially serves our objective: Advances in software research provide the language, model, and analysis to represent and reason about a system's software architecture. Advances in adaptive technology provide mechanisms to enable self-adaptation, in particular to monitor system states and effect changes. Research in related self-healing systems and architecture-based approaches demonstrate point solutions for particular classes of systems and singular quality dimensions.

However, in order to accomplish the thesis objectives enumerated in Section 1.4, we require a self-adaptation framework generally applicable to different styles and quality objectives, the ability to explicitly represent administrator adaptation concepts as operational entities, the mechanism to automatically decide the best course of adaptation, and an integrated approach that saves engineers time and effort. To provide the missing capabilities requires solutions to two problems. The first requires a framework we call "Rainbow" to provide the general supporting mechanisms for self-adaptation. For the second problem, we develop a language called "Stitch," which plugs in to this framework, to provide an appropriate representation for adaptation expertise so that we can reason about and automate adaptation to satisfy multiple objectives.

In this chapter, we provide an overview of the Rainbow architecture-based approach to self-adaptation. We present technical details in later chapters: the Stitch language in Chapter 4, the design of the framework and its customization points in Chapter 5, example instantiations of Rainbow to demonstrate the approach in Chapter 6, and the thesis evaluation in Chapter 7. In the sections that follow, we first enumerate the requirements of a solution, then highlight the Rainbow approach and the framework for realizing the self-adaptation cycle to *monitor*, *detect*, *decide*, and *act*, and conclude with an overview of the process to apply Rainbow.

3.1 Overview of Approach

To overcome the limitations outlined in Section 2.4, we require a principled approach to engineer self-adaptive systems. As noted earlier from the thesis claims, we enumerate three requirements for the solution:

1. Achieve **general** coverage: The approach should apply to a broad spectrum of styles for

typical quality dimensions with which modern architectures are concerned.

2. Facilitate **cost-effective** adaptation engineering: The approach should facilitate low-cost, expeditious efforts to engineer self-adaptation capabilities for supported classes of systems.
3. Enable **transparent** self-adaptation: The approach should allow engineers to separately specify self-adaptation strategies for distinct quality objectives, then integrate them to achieve self-adaptations that trade off across multiple objectives.

Our approach consists of a framework, a language, and an incremental process for engineering self-adaptation. The framework consists of two parts: (1) a generic, reusable infrastructure of common adaptation mechanisms that software engineers can apply to different classes of systems, and (2) a set of explicitly-defined customization points that allow the engineers to *tailor*, or to target and customize, the infrastructure to a specific class of system. The underlying principle here is to separate mechanism from policy, the machinery that performs the self-adaptation work from the instructions that tell the machinery what to do.

The language codifies concepts for defining adaptation policies to achieve multiple, specific adaptation objectives. The incremental process prescribes new roles of *adaptation engineers* (Section 3.5 below) to develop self-adaptation capabilities in the target system, and describes how the capabilities can be developed piecemeal. In particular, the process describes what custom contents to define, which parts to develop, and when and where to define and develop them.

Below, we highlight three important underpinnings of our solution: Software architecture gives us leverage to make self-adaptation general and cost-effective. Control theory provides a well-understood mechanism for closed-loop system adaptation. Utility theory allows decision-making of adaptations that considers multiple factors and is sensitive to its context. We conclude this section with a set of design constraints for a framework that is fit for our adaptation purpose.

3.1.1 Software Architecture Model and Style

The first major underpinning of this approach is the use of a *stylized* software architecture model to monitor and adapt a target system. Like the blueprint of a building, the 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, e.g., 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 perform analyses on the system for such quality attributes as performance, availability, reliability, and security. Together, *vocabulary*, *properties*, *rules*, and *analyses*, summarized below, comprise the building blocks of *architectural style* [AAG95].

1. **Vocabulary** (V) of element types, including component types (e.g., database, client, server, filter, etc.), connector types (e.g., sql, http, rpc, pipe, etc.), 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 or common 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 [SG98] and schedulability analysis for a style oriented toward real-time processing [AVCL02].

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 [MK96] and Wright [All97]), the traditional style lacks natural mechanisms to represent what 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.

In order to handle the notion of dynamism with respect to architectural structure, in this thesis we augment the notion of style with *operators*. We assume that operators are blackbox, target system-provided capabilities defined in a style; hence, we do not address their specification.

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. Style-specific operators are specifiable in terms of generic architectural operators, such as component add or remove, connect, or disconnect, a classification of which is exemplified in [MM03].

As we shall see, 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 classes of system. 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 introduce the framework in Section 3.3, present its design and customization points in Chapter 5, and evaluate how well it satisfies the generality and cost-effectiveness claims in Chapter 7. Next, we discuss control systems, which is integral to the design of our self-adaptation framework.

3.1.2 Control Systems and the Self-Adaptation Cycle

The second major underpinning of this approach is the application of control systems concepts to the adaptation problem. Whereas software systems are traditionally designed as *open-loop* systems, we overcome the challenges of self-adaptation by taking a control systems' view and

closing the loop of control. We further choose a specific type of control system model to make our approach generalizable and reusable across different classes of systems.

In a typical control system, such as the one depicted via *box diagram* in Figure 1.2 on page 4, 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 would require mechanisms to obtain information about the system and its environment as well as to manage the corresponding architecture and environment models. Furthermore, the Controller must be able to select a course of action and effect the changes on the system.

These required capabilities of control correspond to the 4+1 phases of the adaptation cycle, also defined by the IBM Autonomic MAPE Architecture mentioned in Section 1.1.2 [GC03]:

Monitoring phase extracts relevant target system states using techniques of varying intrusiveness, from instrumenting the source code, to intercepting low-level system events, to unobtrusively reading system logs. This phase usually acquires information without interpretation, except as needed to extract that information, such as parsing a log. Due to potential resource overhead, precisely what to monitor is an important consideration.

Detection phase consists of two important steps: (1) interpreting information from the monitoring phase, and (2) deciding whether a problem or opportunity for improvement exists. This phase relies on one or more analytical models to assess conditions of error.

Decision phase comprises two aspects: (optionally) determining the root of the problem, and choosing the appropriate course of remedy. This phase most likely relies on one or more analytical models (as used in *detection*), and requires a repertoire of remedial actions.

Action phase executes the chosen action, effects the appropriate changes on the target system, and may need to recover from intermediate failures. This phase requires some degree of access to the system states, and may need to interpret and handle system-level errors.

Knowledge contains the model(s) and data shared by the separate phases to achieve adaptation.

As we shall see in this chapter, the Rainbow framework realizes these 4+1 self-adaptation phases: *knowledge* is embodied in the architecture model[1] 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 *tactic* and *strategy* as formal concepts of self-adaptation in Chapter 4. Each adaptation decision requires the consideration of multiple factors, which leads to the third underpinning, utility theory.

3.1.3 Utility Theory

The third major underpinning of this approach is the use of utility theory for decision-making that considers multiple factors while being sensitive to the context of use, such as to overall busi-

[1]When we say "architecture model" and, subsequently, "model" in the same paragraph context, we will mean *both* the architecture model and the environment model of the managed system, as described in Section 3.3.1.

ness objectives and priorities. When the self-adaptation mechanism detects an opportunity for improvement in the target system and is choosing a strategy to adapt the system, it must consider numerous factors and make a choice between both similar kinds of strategies and strategies that have juxtaposing effects. Simply stated, the problem is to choose the best strategy to adapt the system, given existing system conditions, that takes multiple objectives into consideration.

To determine which strategy is "best," we need to define values for the objectives, relate the objectives to specific system conditions, and assess the impact of the strategies to the objectives. Since there is uncertainty in the outcome of an adaptation, we also need to estimate the likelihood of observing certain system conditions after executing a strategy. Utility theory, combined with a stochastic model of the strategy outcomes, provides the method to quantify strategies relative to the objectives, under uncertainty. Chapter 4 explains how we leverage utility theory in Stitch.

3.1.4 Design Constraints for Self-Adaptation

We now briefly consider and motivate the design of a generic, self-adaptation framework that addresses our thesis requirements. We explore design alternatives and potential limitation is Chapter 8. To be fit for adaptation purpose, the runtime framework should be designed to monitor the target system dynamically without affecting target-system operation, track its state in a central model to provide overall system context, provide functionalities for closed-loop control, and be flexible to change and resilient to failure. These considerations of adaptation purpose lead naturally to the following design constraints for our solution framework:

- *Asynchronous* monitoring and adaptation from target system operations (cf. Section 8.2)
- *Central* management of system's model (cf. Section 8.1)
- Distinct allocation of *control responsibilities* (cf. Section 8.3)
- *Resiliency* to component failures and eliminating single point-of-failure (cf. Section 8.1)
- Well-defined *customization points* for common functionalities (cf. Sections 5.1 and 8.7)

Next, we introduce a complete example before describing the Rainbow framework.

3.2 Znn.com Example

To illustrate the framework, consider an example news service, *Znn.com*, that serves multimedia news content to its customers, based on 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 Figure 3.1, 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 take advantage of the monitored system and adapt the system to fulfill Znn.com objectives.

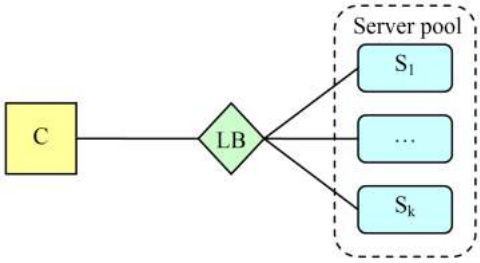


Figure 3.1: Architecture model of the Znn.com system

The business objectives at Znn.com are to 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 minimalist 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 would switch 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 not only on the conditions of the system, but also 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 vs. 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 for potentially conflicting qualities such as performance and security. This work is an important step in that direction: to allow automation of adaptations that must strike a balance between multiple objectives (we will address this issue in Chapter 4).

In the Rainbow framework, the adaptation mechanism uses the architecture model of the system to monitor the system and reason about appropriate strategies. Abstractly speaking, the framework shown in Figure 3.2 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

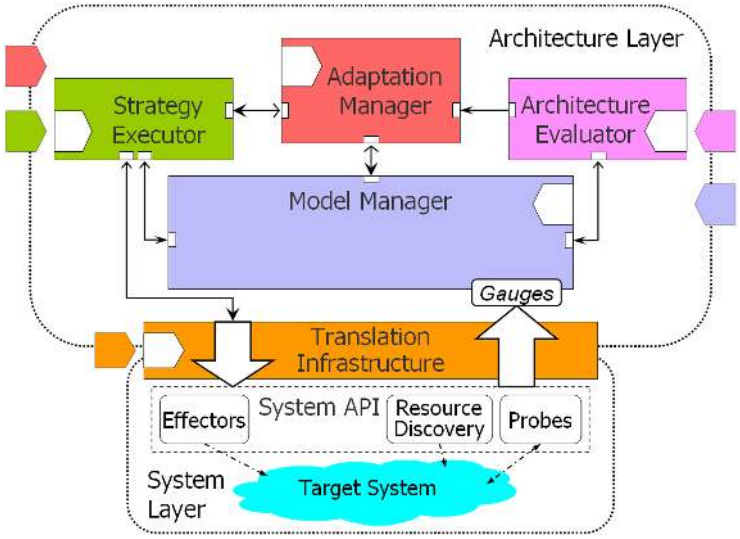


Figure 3.2: The *Rainbow framework* with notional customization points

determined by the architectural constraints. If the evaluation determines that the system is not operating within the accepted range, the Evaluator triggers the *Adaptation Manager* to initiate the adaptation process and choose the strategy. The *Strategy Executor* executes the strategy on the running system via system-level *effectors*.

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 as needed 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 customized 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 infrastructure and developed an iterative process to facilitate reuse of self-adaptive functionalities and reduce the cost and effort of achieving self-adaptation.

3.3 Tailorable Rainbow Framework

To fulfill the self-adaptation requirements outlined in Section 3.1, we envision a framework with general and reusable infrastructures 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 Figure 3.2. The *Rainbow framework* consists of a number of components that provide the monitoring, detection, decision, and action capabilities of self-adaptation. As indicated below, several of these capabilities derive from prior work, while the decision and

language-associated mechanisms are new to this thesis. Rainbow functionalities are described abstractly in this section, and the framework design is presented in detail in Chapter 5.

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, as described in Chapter 4, a modular adaptation language to specify the adaptation policy allows engineers to consider adaptation concerns separately and then compose them. In short, assessed abstractly, the Rainbow framework has the potential to satisfy the generality, cost-effectiveness, and transparency requirements of this thesis (Section 1.4).

3.3.1 Rainbow Models

The Rainbow framework leverages two kinds of models, the architecture and the environment, to make adaptation decisions. An architecture model reflects abstract, runtime states of the target system itself. While existing approaches in architecture-based self-adaptation share this feature, as mentioned in Section 1.3.1, current approaches fail to take advantage of 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 its executing environment and the resources used. For example, when additional servers are needed, the environment model indicates what spare servers are available. When a better connection is required, the environment model contains information about the available bandwidth of other communication paths.

```
1  Family ClientServerFam = {
2      Component Type ClientT = { ... }
3      Component Type ServerT = { ... }
4      /* ... connector/port/role/property definitions ... */
5      Property Type LatencyPropT = float;
6      Property MAX_RESPTIME : float;
7  }
8  System ZnnCSSystem : ClientServerFam = {
9      Property MAX_RESPTIME : float = 1000.0;
10     ...
11     Component Client1 : ClientT = new ClientT extended with { ...
12         Property avg_latency : LatencyPropT = 0.0
13         invariant self.avg_latency < MAX_RESPTIME;
14     }
15     Component Server1 : ServerT = new ServerT extended with { ... }
16 }
```

Figure 3.3: Example definition of the architecture model in Acme

Focusing first on the architecture model, managing an executing system dynamically requires knowing what entities are present, what runtime states they are in, and how they communicate. 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 [CBB+03] in

Acme. 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. Figure 3.3 illustrates a partial architecture description of an example style, ClientServerFam, with component, connector, and property types. Instantiated from that style is a partial Znn.com system description, ZnnCSSystem, which defines a client instance with an *average_latency* property value (line 12) and an architectural constraint (line 13), which we describe in Section 3.3.4 on the Architecture Evaluator.

```
1  Family EnvType = {
2      Property Type ResourceStatePropT =
3          Record [ unit : string; total : float; available : float; used : float; ];
4      ...
5      Component Type NodeT = {
6          Property cpuOverall : ResourceStatePropT;
7          Property memoryMain : ResourceStatePropT;
8          Property storageLocal : ResourceStatePropT;
9          Property socketPool : ResourceStatePropT;
10         Property batteryOverall : ResourceStatePropT;
11     }
12     Connector Type EdgeT = {
13         Property bandwidthAvg : ResourceStatePropT;
14         Property capacity : ResourceStatePropT;
15     }
16     Port Type NetworkPortT = { }  // ... and Role Type NetworkRoleT
17     // connector-port-role to capture containment relationship
18     Connector Type MappingT = { }
19     Port Type ContainmentPortT = { }  // ... and Role Type ContainerRoleT
20     Port Type PartPortT = { }  // ... and Role Type PartRoleT
21 }
22 System ZnnEnvModel : EnvType = {
23     Component elementX : NodeT = new NodeT extended with { ... }
24     ...
25 }
```

Figure 3.4: Example definition of the environment model in Acme

Turning now to the second model, the environment model captures states of the target system's execution environment to provide additional information for the self-adaptation process. Two central modeling questions are: (1) what environment information should be captured, and (2) how to capture it? To support self-adaptation, of primary concern are the *kind*, and *usage status* of environment resources. Examples include CPU load and free memory on a host machine, total capacity and available bandwidth of connections between nodes, and aggregate entities such as particular types of computing nodes, applications, and services. The broad notion of *resources* includes various granularity of resources necessary to facilitate reasoning about adaptation.

To capture environment information, as with architecture, we take a graph approach to represent resources as nodes and typed relations as edges (physical connection, containment, and dependencies). In general, there are resource (node) types and relation (edge) types, which we capture in an *environment style* and instantiate for a specific system environment. We have explored one style of environment with performance-centric resources, as illustrated in Figure 3.4, but other styles of environment may be defined, such as an environment that captures sources of attack to help reason about adaptation for security concerns. The necessary environment information typically relates closely to the system elements. Thus, we maintain a mapping between

architecture-model elements and environment-model elements. To manage scope, we use the graphical notion of *hop count* [Uni96] as a model parameter and track only the environment elements within a defined number of hops from the core, architectural elements.

Because Acme has explicit constructs corresponding to nodes and edges, supports types, and has the extensible *property* construct, we model the environment in Acme, which also allows us to leverage existing tooling support for model description. Figure 3.4 illustrates a partial environment description for Znn.com, which we discuss in more detail in Chapter 5.

3.3.2 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, we need mechanisms that hook into the target system and understand what is represented in the model. The layer marked *Translation Infrastructure* in Figure 3.2 provides these **monitoring** and **action** (cf., Section 3.1.2) 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 [CHG+04, GSC01, Bal01].

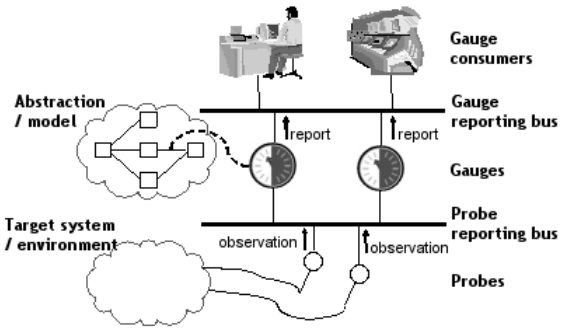


Figure 3.5: Monitoring mechanisms: probes and gauges

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 Figure 3.5, 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).

To tailor the monitoring mechanisms, an adaptation engineer identifies the properties of spe-

cific element types to monitor[2] 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. Table 3.1 illustrates a gauge instance specification that defines an instance G_L of a predefined latency gauge type Latency_Gauge_T, shows the architecture model and property (L.Latency, where L is a connector instance in the model) associated with the gauge (Latency), and indicates the target probe type (Target_Probe="pingrtt") to which the gauge maps. While we require probes and gauges to enable overall Rainbow functionality, they are not the focus of this thesis. Chapter 5 describes how we use probes and gauges and how we associate gauges with the architecture model.

Table 3.1: An example gauge instance specification

Gauge Name : Gauge Type	G_L: Latency_Gauge_T
Model Name : Model Type	ZnnSys : Acme
Mapping	<Latency, L.Latency>
Setup Values	Src_IP_Addr : String = L.IP1; Dst_IP_Addr : String = L.IP2;
Configuration Values	Sampling_Freq : int = 100; Target_Probe: string = "pingrtt";

Action Mechanisms *Effectors* carry out change operations on the target system; they are associated with architectural operators in the Rainbow Architecture Layer (Figure 3.2). 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 [VKK01]). As with probes and gauges, we require effectors to enable overall Rainbow functionality, but they are not the focus of this thesis. Rainbow's dependency on monitoring and action capabilities for the target system is not a serious limitation. We build on others' work on probes and effectors, including adaptive middleware technology [CDS01, ACM02]. Furthermore, modern systems increasingly support probing and effecting capabilities, as evidenced in industrial products such as from IBM's Autonomic Computing [GC03] and Microsoft's Dynamic Systems Initiatives [Mic03].

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 prior publication [CHG+04], we identified four distinct kinds of correspondence mappings, maintained by the Translation Infrastructure, to facilitate translation of control[3] 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 and illustrate them in Table 3.2:

Type 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.

[2]Note that *element types* and *properties* correspond to sets V and P defined for style in Section 3.1.1.

[3]In this dissertation, monitoring mechanisms do not currently use the translation mappings.

Element An element map relates an element instance in the architecture model with an entity in the target system, including the property values.

Operation An operation map relates an architectural operator, along with its formal parameters (type and name), to an effector operation, along with the corresponding parameters.

Error An error map relates the identifier and error sources of an exception in the target system to a corresponding error at the architecture-level.

Table 3.2: Summary of four Translation Infrastructure correspondence mappings

Map	Description	Definition	Example Mapping Instance
Type	Element kind & properties	*archType* ($prop_1$,.., $prop_n$) :: *sysType* ($prop_1$,.., $prop_n$)	GatewayT (location:String, cost:float) :: ServiceGW (ip:InetAddr, cost:double)
Element	Model or system entity	*archInst* (*type*, $prop_1$,..,$prop_n$) :: *sysEnt* (*type*, $prop_1$,..,$prop_n$)	g1 (GatewayT, location="PA", cost=1.3) :: sg1 (ServiceGW, ip="10.1.2.3", cost=1.3)
Operation	Action & parameters	*archOp* ($param_1$,.., $param_n$) :: *sysOp* ($param_1$,.., $param_n$)	startSvc (s:GatewayT, timeout:float) :: Service.start (s:ServiceGW, timeout:double)
Error	Operation problem	archErr :: sysEx	GatewayNotFound :: GatewayHostNotFoundEx

3.3.3 Model Manager

We need a mechanism that updates and controls access to the architecture model, the shared **knowledge** among the adaptation mechanisms (cf., Section 3.1.2). The *Model Manager* manages both the architecture and environment models of the target system. It leverages prior work on architecture modeling, particularly the ADL called Acme and the supporting typechecking software library [GMW00]. It maintains references between elements of the environment and the architecture models. It tracks the model states, maintains correspondence of the models to system and environment states via gauges, provides the Rainbow components with shared access of 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 (cf., Section 3.5) specifies a vocabulary (a family of element types in Acme) 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.[4]

3.3.4 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., Section 3.1.2). When any model property

[4]Note that *vocabulary* and *properties* correspond to sets V and P defined for style in Section 3.1.1.

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 (Figure 3.2) to trigger adaptation. This mechanism leverages prior work on the use of architectural (Acme) constraints, specified in first-order predicate logic, to identify flaws in system design [Mon99]. We extend this work by checking architectural constraints over runtime system properties to detect target system problems at run time.

To tailor the Evaluator, a style writer (cf., Section 3.5) specifies as rules the topological and behavioral constraints[5] 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. A simple constraint is illustrated on line 13 in Figure 3.3, requiring that the average latency experienced by the client component never exceed a "max_latency" threshold.

3.3.5 Adaptation Manager

Once a problem is detected, we need a mechanism to **decide** on the appropriate adaptation remedy (cf., Section 3.1.2). When triggered by the Architecture Evaluator, the *Adaptation Manager* uses the architecture model to select a course of remedial strategy that best suits the present problem state of the system, then coordinates the execution of that strategy. The Adaptation Manager combines utility, decision, and control theories to solve the decision-making problem in self-adaptive systems.

In addition to the *operator*, this thesis formalizes the notions of *strategy* and *tactic* as core concepts for decision-making in the Stitch self-adaptation language, detailed in Chapter 4. Briefly, a *tactic* defines an action, packaged as a sequence of commands; it specifies conditions of applicability and expected effects. A strategy captures 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. A strategy also specifies conditions of applicability that determine in what contexts it should be involved.

During the adaptation process, the Adaptation Manager selects the best strategy given observed system conditions. It first scans a repertoire of strategies for applicable ones based on present system conditions, then scores those strategies based on their expected utilities relative to a defined set of quality objectives, and finally selects the highest-scoring strategy to execute.

To tailor the Adaptation Manager, the engineer specifies a set of adaptation strategies and the policy for selecting strategies. However, a number of critical issues remain: How should strategies and selection policies be specified? How is the appropriate strategy determined? What does one do if conflicts arise? We discuss these issues further in Chapter 4.

3.3.6 Strategy Executor

Once a strategy is chosen, we need a mechanism that understands how to interact with **action** hooks in the target system to carry out the adaptation. The *Strategy Executor* is dispatched by the Adaptation Manager to execute the selected strategy on the target system. It resolves model

[5] *Constraints* correspond to set R defined for style in Section 3.1.1.

references within the strategy against the Rainbow model, observes model states and evaluates branch conditions to determine tactics to execute, maps operators within tactics to system-level effectors to carry out changes, and handles errors from effector invocation.

The Strategy Executor is tailored by the set of operators[6] that are defined as part of the system's architectural style. Examples of operators from a few architectural styles include:

pipe-filter style: add- / remove- / replaceFilter and connect- / disconnectPipe

service-coalition style: add- / remove- / start- / stopService and connect- / disconnectProtocolX

3.4 Rainbow Application to Znn.com

To illustrate the Rainbow framework, let us walk through the Znn.com example. Table 3.3 highlights how each of the Rainbow components is customized for Znn.com. This example is simplified to illustrate only the major features of Rainbow. Detailed language features, including strategy selection and failure handling, appear in Chapter 4; Chapter 5 details Rainbow customization; and Section 6.5 describes the full-fledged Znn.com example.

Three quality objectives are defined: timely response, high-quality content, and low-provision cost. 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. 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. Three tactics are defined, along with their expected cost-benefit impact on the quality objectives: to switch server content to from multimedia to textual and vice versa, and to adjust the server pool size by i servers. Lastly, two strategies are defined (using the tactics above): one to reduce the request-response time and one to improve content fidelity.

The customized Rainbow framework for Znn.com works as follows. 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, selects

[6] *Operators* correspond to the set O defined for style in Section 3.1.1.

Table 3.3: 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
Tactic	Adaptation Manager	switchToTextual [$-uR$, $-uF$, $0uC$], switchToMultimedia [$+uR$, $+uF$, $0uC$], adjustServerPoolSize(i) [$-iuR$, $0uF$, $+iuC$]
Strategy	Adaptation Manager	simpleReduceResponseTime, improveFidelity

the highest-scoring one, and delegates it to the Strategy Executor. In this case, simpleReduceResponseTime might be selected, which either activates more servers or decreases content quality, depending on specific system conditions. Assuming conditions favor decreasing content quality, the Executor evaluates the branches of simpleReduceResponseTime, chooses to execute the tactic switchToTextual, and invokes the setFidelity operator, which 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.

3.5 Adaptation Engineering Process

The Rainbow framework not only provides customizable components, but also allows engineers to focus on adaptation-level concerns by enriching the adaptation design process with abstract adaptation concepts like strategies, tactics, operators, effectors, gauges, and probes. Associated with these concepts and the framework components, we have developed an iterative process to engineer a system for self-adaptation. The aim of the process is to produce self-adaptive systems. Integral to this process is a team of *adaptation engineers* who perform various adaptation engineering roles—*adaptation integrator*, *system adapter*, *gauge writer*, *style writer*, *tactic writer*, and *strategy writer*—to evolve and augment a target system with self-adaptation capabilities.

The process is focused primarily on gathering the necessary artifacts to customize the Rainbow framework, as illustrated briefly in Section 3.4 and detailed in Chapter 5. First, an adaptation integrator queries the system owners for the business objectives to guide system self-adaptation (e.g., "timely response," "high-quality content," "low provisioning cost"). The adaptation integrator will later complete the customization of the self-adaptation framework. As we will show in Chapter 4, the business objectives, in the form of utility functions and preferences over quality dimensions, are critical for the decision-making process of self-adaptation.

Next, the adaptation integrator studies the target system to determine its architectural style (e.g., N-tier) and to find an existing definition from a style library. If no existing style can be

reused, then a style writer would need to define a new style. Finally, the adaptation integrator adapts the style to fit the target system and to produce a C&C architecture model (e.g., ZnnCSSystem). Access to architectural documentations from the software lifecycle of the target system may expedite this modeling effort. In practice, style development most likely proceeds incrementally as a combination of using an existing style (e.g., ClientServerFam) and refining it to fit a new system.

Based on the quality objectives, the system adapter identifies pertinent system properties for which to develop and deploy probes (e.g., ClientT.reqRespLatency and ServerT.fidelity), and changeable states for effectors (e.g., changeFidelity.pl). Depending on the fit of the style, the adaptation integrator adds constraints to the model (e.g., the invariant on line 13 of Figure 3.3) to adapt for the objectives. In addition, the gauge writer finds or develops gauges to update the architectural properties necessary for adaptation, and the adaptation integrator maps gauges to corresponding probes. The adaptation integrator also establishes mappings between operators and effectors, as discussed in Section 5.1.6.

With the help of the target system's administrators, tactic and strategy writers then develop *tactics* and *strategies*, which emulate how the sys-admins would adapt the system. Since tactics and strategies form the decision-making elements, factors for decision also need to be described. The adaptation integrator makes initial approximations of how each of the tactics impacts the quality dimensions and develops relative weights for strategy selection using the elicited preferences over the same dimensions. We focus on these decision-making details in the next chapter.

Finally, the adaptation integrator hooks up these parts through the Rainbow framework, and tests the roundtrip adaptation. Roundtrip testing is the integration test, and thus a crucial step, of adaptation engineering. A very important feature of this process is the incremental development and integration of adaptation functions, allowing iterative realization of overall system self-adaptation capabilities. Undoubtedly, some upfront cost is required to develop the first architecture styles, gauges, effectors, etc. However, styles, gauges, probes, and effectors are reusable artifacts; thus, building libraries of them will amortize cost and reduce adaptation engineering efforts for future projects. Furthermore, strategies and tactics can be reused in systems with similar styles and concerns. We discuss reuse further in Section 7.2.

3.6 Summary

In this chapter, we reiterated the requirements of **generality**, **cost-effectiveness**, and **transparency** to engineer architecture-based self-adaptive systems. We highlighted software architecture, control theory, and utility theory as three important components of our approach. We introduced a two-part Rainbow approach and framework for realizing the self-adaptation cycle, which consists of generic infrastructures with customizable elements to enable monitoring, detection, decision, and action. We illustrated the approach with the Znn.com example. Finally, we described the iterative and incremental Rainbow adaptation engineering process. Next, we present the language for self-adaptation.

Chapter 4

Stitch Self-Adaptation Language

This thesis aims to provide a cost-effective engineering approach to enable self-adaptation capabilities, focusing specifically on automating mundane and routine system administration tasks. Automating self-adaptation can reduce IT operation cost for many businesses, as argued in Section 1.1. In the previous chapter, we provided an overview of Rainbow as an integrated framework for self-adaptation that applies generally to different styles and quality objectives, and can potentially save engineers time and effort adding and evolving adaptation capabilities to their systems. To enable architecture-based self-adaptation, not only do we need the mechanisms to monitor system states, detect problems, and effect adaptation changes, we also need a way to capture routine human adaptation knowledge as explicit adaptation policies. By capturing *what* to adapt for, *when* to adapt, and *how* to adapt, these policies provide the Rainbow mechanisms instructions for automating adaptations.

In this chapter, we enumerate the requirements for an expressive language of self-adaptation, in the context of Rainbow, and as motivated by the need to automate routine system administration tasks. We present a language, called *Stitch*, which embodies the adaptation concepts of *operator*, *tactic*, and *strategy* as first-class entities to capture a (a) system-provided command, (b) single adaptation step with cost and benefit impact, and (c) packaged pattern of adaptation steps. Stitch supports explicit representation of business objectives and relation of tactic costs and benefits to objectives, reusable strategies of self-adaptation, a utility-based algorithm for strategy selection, leverage of architectural style to support a broad spectrum of systems, the ability to control timing, and the handling of uncertainty. Using the Znn.com example, we illustrate the expressiveness of this language with respect to representing adaptation expertise and supporting multi-objective trade-offs.

4.1 Rainbow Context for Language

While describing the overall Rainbow approach in Chapter 3, we alluded to features of a *self-adaptation language* and briefly illustrated them using Znn.com in Section 3.4. To motivate the requirements for this language, we first examine its role in the context of the overall framework with respect to the adaptation process. Specifically, we consider what are the inputs and outputs when the adaptation mechanism executes adaptations expressed in this language.

Revisiting the Znn.com example, when a client experiences above-threshold latencies, violation of the request-response constraint triggers the Adaptation Manager to initiate the adaptation process, using information available in the architecture and environment models. Effectively, the current system and environment conditions, as observed through Rainbow's models, are used to filter out the inapplicable strategies and to score the applicable subset. The highest-scoring strategy is given to the Strategy Executor to be carried out on the target system.

The adaptation process requires both static and dynamic inputs to the mechanism. The static inputs consist of the architectural style of the target system, a repertoire of strategies, and a utility profile specification. The dynamic inputs include the architecture and environment model instances and a trigger caused by one or more violated constraints.

To carry out adaptations, the framework provides a number of capabilities for evaluating adaptation instructions. The *Model Manager* allows querying the elements and properties of the architecture and environment models. For example, an expression written in the language can query the Model Manager for the current value of the (gauge-updated) load property of a server instance. The *Strategy Executor* provides access to the architectural operators defined by the style. For instance, a statement written in the language can directly invoke the architectural operator, startService to start an instance of a Service component in the target system (which is translated to the equivalent system-level operations). Finally, the framework supplies various library utilities, such as set operations, mathematical functions, and I/O operations.

4.2 Requirements for the Self-Adaptation Language

To automate adaptation tasks, we require a language sufficiently expressive to represent routine human expertise, while flexible and robust enough to capture complex preferences. At an abstract level the language will need to define operational self-adaptation *concepts* and support the specification of a *value system* to enable *choice* of adaptation:

Concepts Define concepts that formalize the operational aspects of self-adaptation: the language should allow one to express both low-level operations and high-level abstractions.

Value System Provide a way to specify value systems for comparing adaptations: the language should enable representing quality objectives, preferences across the objectives, and the impact of adaptations on those objectives.

Choice Apply the value system to select the best course of adaptation action: the language should allow analyzing the best course of action given the specified objectives, preferences, and adaptations.

As noted earlier, our approach to addressing the general problem of automated self-adaptation is to focus on the specific domain of system administration. Hence, to determine the required features of Stitch, we first examine the nature of system administration tasks.

4.2.1 Nature of System Administration Tasks

Imagine a sys-admin, Sam, who manages the Znn.com infrastructure. We consider the task, knowledge, and cognitive model involved for Sam to keep Znn.com operational. Under normal

conditions, faced with the large number of system conditions to check, Sam chooses to monitor a system property that has historically been a good indicator of problems, i.e., the average request-response time. In general, the sys-admin monitors a few top indicators of opportunities for improvement. The language should *support capturing adaptation conditions*.

To enable administration, the system generally provides a suite of basic operators to change system states, such as starting an application, killing a process, or invoking specific system utility commands. The language should *take advantage of system-provided operators*.

Although Sam might directly use a subset of the given operators in isolation, Sam typically builds larger-unit actions to facilitate system adaptation. To form these reusable units of actions or *tactics*, Sam has packaged frequently-performed sequences of operations into scripts, such as instantiating a new web server or changing the content quality of active servers. Effectively, the sys-admin has built up an arsenal of tactics to use at each adaptation step. Sam knows from experience when to apply each tactic, what its costs and benefits are, and what outcome to expect. Sam also treats each tactic as an abstract *step* of adaptation that runs to completion. The language should *support the definition of tactics*.

In general, solving a problem may require more than one of these actions, with decision points along the way involving intermediate observations of the system. For example, in response to a problem, Sam typically follows a strategy from experience and carries out a number of adaptation steps, with the aim of achieving a favorable aggregate impact over the whole path. In some cases, this may involve putting the system temporarily in a state that provides fewer services (e.g., rebooting a few servers). At each step in the strategy, Sam observes system conditions to take a paired action; Sam may also decide to do nothing, or to complete or abort the strategy. Because the exact outcome of an action is uncertain, after taking each action, Sam may wait for it to take effect before taking the next step. The language should *support the definition of strategies composed of steps of condition-action-delay*.

Assume that at some point Sam notices the average request-response time at Znn.com rising above a maximum threshold. From the system state, Sam realizes that this rise is attributable to either a performance or a security problem, but decides from observing the system conditions to pursue a performance-improvement strategy. The language should *support specifying conditions of applicability for strategies*.

But how does Sam know how to pick a strategy from among applicable ones? In the case of systems like Znn.com, Sam manages the system to satisfy three quality objectives of concern to the business, each objective signifying a dimension: (a) response time experienced by the customers, (b) news content quality, and (c) server provisioning cost to Znn.com. In general, each dimension has a measurable system property, and there might be a number of potentially conflicting objectives. To facilitate resolving such conflicts, the business has notions of utility for, and preferences across, the quality dimensions. The language should *support characterizing quality dimensions and utility preferences across the dimensions*. Sam's choice of adaptation strategy must make an appropriate trade-off across the dimensions based on the utility preferences of the business. The language should *support utility-based strategy selection.*

For example, there might be several performance-improvement strategies, including one based on server load, another based on content fidelity, a smart one incorporating both, and a sophisticated one that rejects a subset of requests. Before pursuing a particular strategy, Sam considers several factors, including how many resources an action might require, how long the

action might take, and how much the action might improve the system. The language should *support characterizing cost-benefit attributes*.

Continuing the example, with the smart performance-improvement strategy, Sam checks server loads, observes high load, and concludes that there is a popular news event. Sam invokes a script to enlist a free server into the server pool, then waits a few seconds to see if that reduces the system load. Sam observes that the load is still high and invokes another script to switch all the servers to textual mode so that the system can recover and fulfill pending customer news requests. After waiting a little while, Sam observes that system load has improved and the strategy has completed.

As another example, after switching all the servers to textual mode, Sam considers the size of the server pool and the company operating budget to determine whether to increase the pool size to serve more requests. With this increase in pool capacity, it turns out that Sam can switch the servers back to multimedia mode. This case exemplifies an adaptation where Sam has identified an opportunity for improvement rather than a problem to fix.

We summarize the concepts of self-adaptation from this analysis in Table 4.1 below.

4.2.2 Language Design Considerations

In addition to examining the nature of system administration tasks, we must also consider issues in specifying adaptation expertise during the engineering phase and in executing adaptations during the operation phase. These issues motivated the design of various constructs in the Stitch language, which we present in the next section.

Designing self-adaptation capabilities for a target system typically requires considering multiple quality attributes, which depend potentially on different domain expertise and, thus, multiple experts. An essential feature for a cost-effective approach to engineering self-adaptation is to enable both *per-concern* knowledge capture and *cross-concern* adaptation engineering. We want to enable different experts to capture their domain expertise individually for their particular concerns (strategy conditions of applicability). In a separate phase, we want to be able to compose and integrate expertise from domains relevant to the quality attributes of the target system. This engineering scheme allows us to build a library of adaptation expertise incrementally and to reuse expertise from system to system.

During adaptation operation, due to asynchrony of adaptation (cf. Section 8.2), there is often uncertainty in (a) whether an observed problem condition is transient and (b) when an adaptation action has taken effect in the target system. To cope with *condition uncertainty*, an effective technique is to capture the *conditions of applicability* for an adaptation action and then verify that condition at the moment of execution. To cope with *effect uncertainty*, an effective technique is to designate a *timing delay* for the adaptation mechanism to wait and observe the effect.

As we present in the next section and discuss further in Section 8.4, we need three distinct constructs of operator, tactic, and strategy for separation of concerns. Tactic-tactic and strategy-strategy invocations complicate analysis, but loops within these constructs allow for *repetition* and *reuse* of adaptation logic. Finally, restricting utility selection to only the strategy level allows for adaptation steps that *temporarily reduce target-system utility* (e.g., reboot).

In fulfillment of the above requirements, we now present Stitch, the self-adaptation language.

Table 4.1: Stitch self-adaptation concepts motivated by the sys-admin's tasks

Self-adaptation concept	Sys-admin task, knowledge, model
operator	Single command in system, e.g., kill process
tactic	Script of commands with conditions + effects, e.g., addWebServer
cost-benefit attributes	Factors considered in strategy choice w.r.t quality dimensions
strategy	Pattern of adaptations with condition, action, and delay
conditions of applicability	System conditions to decide what strategies are applicable
quality dimensions	Business qualities of concern
utility preferences	Business preferences over quality dimensions
adaptation conditions	System indicators of opportunities for improvement
strategy selection	Decision between strategy alternatives w.r.t. quality dimensions

4.3 Self-Adaptation Concepts of Stitch

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. In this section, we show how Stitch captures these three kinds of information and describe its execution context.

4.3.1 Overview

Based on our analysis of the sys-admin's task, knowledge, and cognitive model, we can generalize a number of core concepts for automated system self-adaptation, summarized in Table 4.1: *operator*, *tactic* with *cost-benefit attributes*, *strategy* with *conditions of applicability*, *quality dimensions*, *utility preferences*, *adaptation conditions*, and *strategy selection*. In particular, we identify the following list of language features:

1. *Operator*: An operator is a basic command provided by the target system. E.g., operator stopService provided by the service-coalition style.
2. *Tactic*: A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effects, and (in a separate customization step) *cost-benefit attributes* to relate its impact on the quality dimensions. E.g., switchToTextualMode, as shown later in Figure 4.2.
3. *Strategy*: A strategy captures a pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action (tactic), possibly waiting for the action to take effect. A strategy specifies conditions of applicability that determine in what contexts it should be involved. E.g., SimpleReduceResponseTime, as shown later in Figure 4.3.
4. *Quality dimensions*: A quality dimension characterizes a business quality of concern as a utility functions and maps it to a monitored architectural property. E.g., Average response time (uR) is mapped to ClientT.experRespTime in the architecture and has the utility function defined by the points $\langle (0,1), (500,.9), (1500,.5), (4000,0) \rangle$ to represent the utility of average response time at 0, 500, 1500, and 4000 ms.

5. *Utility preferences*: Utility preferences capture business preferences over the quality dimensions. E.g., $[w_1 : 0.6, w_2 : 0.3, w_3 : 0.1]$ might represent preferences where dimension u_1 is twice as important as u_2, and u_2 is three times as important as u_3.
6. *Adaptation conditions*: Adaptation conditions identify opportunities for improving the system. E.g., invariant self.avg_latency < MAX_RESPTIME.

Revisiting the top-level language requirements on page 40, note that features 1–3 fulfill the operational *concepts* requirement and features 4–6, the *value system* requirement. The *choice* requirement is fulfilled by a strategy selection process that we discuss in Section 4.3.6. We now highlight how the Stitch concepts express the important aspects of the Rainbow adaptation process to enable automated system self-adaptation. Flowcharts of the adaptation process (Figure 4.4), strategy execution (Figure 4.5), and tactic execution (Figure 4.6) are shown on pages 64–65.

1. When the *Architecture Evaluator* detects an *adaptation condition*, it triggers the *Adaptation Manager* to initiate a round of adaptation.
2. 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), then selects the best strategy from the subset by computing the expected[1] utility of each strategy as follows (Section 4.4.2 explores the selection semantics):
   (a) Compute the expected aggregate impact of each strategy on each *quality dimension* using the *cost-benefit attributes* specified for the tactics;
   (b) Score the strategies using the *utility preferences* over the quality dimensions; and
   (c) Select the highest-scoring strategy.
3. The *Strategy Executor* evaluates the chosen strategy, which can be understood as a tree in which nodes represent actions; edges, conditions; and the tree root, the initial node.
   (a) The conditions preceding each tree level are evaluated against the model to determine which branch applies. If more than one applies, a branch is chosen randomly.
   (b) The *tactic* of the associated branch is executed, and its *operators* effected on the target system via effectors.
   (c) At the end of the tactic, the Strategy Executor observes the system (reflected in the model) for the effect of the tactic to be achieved, up to a specified window of delay.
   (d) Then the conditions of the next tree level in the strategy are evaluated, again, possibly referencing the updated model.
   (e) This continues until (i) **done** is reached, which terminates the strategy successfully or (ii) **fail** is reached, which aborts the strategy and notifies a Rainbow administrator.

We now describe how the Stitch features are specified. We first present the value system features, followed by the operational concepts, and finally the choice process.

[1] As we shall see in Section 4.3.5, there are uncertainties in the outcome of the tactic, so the computed cost-benefit impact and utility are "expected" in the probabilistic sense.

4.3.2 Quality Dimensions, Utility Preferences, and Adaptation Conditions

As noted above, *quality dimensions* determine *what* to adapt for and correspond to the business qualities of concern for the target system, e.g., concerns of system reliability, service availability, or database performance. A quality dimension provides a notion of utility for particular values of a quality attribute. Hence, each dimension defines a utility function and maps to an architectural property monitored by Rainbow. As described below for the *tactic*, we quantify the impact of a strategy on each of the quality dimensions to determine the merits of one strategy over another.

Table 4.2: Data schema for a Utility Profile

Field	Description	Definition	Example
identifier	A unique mnemonic	string	"uR"
label	Human-readable name	string	"Average Response Time"
description	Descriptive comment	string	"R, client experienced response time (ms), float arch property"
mapping	Monitored arch property	string	"ClientT.experRespTime"
utility function	Type & domain-range	linear | sigmoid | custom; $\langle (x_1, y_1), .., (x_n, y_n) \rangle$	custom; $\langle (0, 1), (500, .9), (1500, .5), (4000, 0) \rangle$

Each quality dimension is captured as a Utility Profile, its data schema summarized in Table 4.2. The profile consists of an identifier, label, description, mapping to a monitored architectural property, and utility function definition. The mapping refers to a property defined in the architectural style, and allows designating a smoothing function (e.g., exponential average) for accumulating the series of property values. The utility function could be captured as a linear or sigmoid function with two defining points, or an explicit set of value pairs (with intermediate points linearly extrapolated).[2] An example profile written in YAML [YAM04],[3] from Section 5.1.5, appears below:

```
1  utilities:
2    uR:
3      label: "Average Response Time"
4      mapping: "[EAvg]ClientT.experRespTime"
5      description: "R, client experienced response time (ms), float arch property"
6      utility:
7        0: 1.00
8        500: 0.90
9        1500: 0.50
10       4000: 0.00
```

Utility Preferences *Utility preferences* define the relative importance between the quality dimensions. Given a set of dimensions, it may be impossible to achieve all of them optimally due to resource constraints (e.g., there is only sufficient bandwidth to fulfill objective X or Y, but not both) and fundamental conflicts between certain quality attributes, such as performance and security (e.g., turning on intrusion detection reduces risk, but increases service latency). However,

[2] In the dissertation (see Chapter 8), we currently support only the latter.

[3] YAML is a standard, human-readable, indentation-sensitive, structured data format that is expeditious to convert into Java collection objects and to translate into XML.

it is often possible to assign different levels of importance to the dimensions (e.g., X is valued to be twice as important as Y). Utility theory provides a systematic technique to do so.

From utility theory [Wik08e], a von Neumann-Morgenstern utility function $u_d : X_d \rightarrow \mathbb{R}$ assigns a real number to each quality dimension d, which we can normalize to the range $[0, 1]$. Across multiple dimensions, we can attribute a percentage weight to each dimension to account for its relative importance compared to other dimensions. These weights form the utility preferences. The overall utility is then given by the utility preference function, $U = \sum_d w_d u_d$. For example, if three objectives, u_1, u_2, u_3, are given decreasing importance as follows: the first is twice as important as the second, and the second is three times as important as the third. Then the business utility preferences would be quantified as $[w_1 : 0.6, w_2 : 0.3, w_3 : 0.1]$.

Adaptation Conditions As noted above, *adaptation conditions* indicate opportunities for improving the target system and determine *when* to adapt. To automate system self-adaptation, we define quantitative expressions with respect to specific quality dimensions, similar to a service-level contract of an electronic stock exchange stipulating that premium trades must clear within 2 seconds of execution. After eliciting quality dimensions from the system owner, the adaptation engineer identifies specific system properties that correspond to these dimensions and determine measurable threshold quantities to define quality-of-service statements as adaptation conditions. We specify, in the architecture model (cf. lines 9 and 13 in Figure 3.3), a measurable threshold quantity as a system-instance architectural property and an adaptation condition as an architectural constraint, a violation of which identifies an opportunity for adaptation:

```
1  // threshold quantity (property defined in the style, value in the instance)
2      Property MAX_RESPTIME : float = 1000.0;
3  // adaptation condition (defined in the style)
4      invariant self.avg_latency < MAX_RESPTIME;
```

4.3.3 Operator

An *operator* represents a basic command provided by the target system; it corresponds to a system-level effector as a primitive building block.[4] The *style writer* (see Section 3.5) defines the operators for a given style to specify the available changes on systems in that style. For example, an architectural operator to stop a service, stopService, might translate to a system-level effector that terminates the running service process. Other examples of operators are startServer, setFidelity, and connectPipe.

Operators are not specified as part of Stitch, but rather, they are specified as part of the style (not addressed by this thesis) and mapped to effectors (see Section 5.1.6). In Stitch, the *operator* manifests itself as an identifier (the *identifierPrimary* term in Figure 4.1, line 20). In Figure 4.2, the import statement on line 3 imports the operators provided by the Znn.com architectural style, and the tactic statement on line 12 shows invocation of an operator. Note that operators may only be invoked from within the action block of a *tactic* (via the *statement* form).

[4] In practice, the operator, though having a simple interface, may require a complex implementation.

```
1  # Note that the grammar, written in ANTLR, uses the following BNF conventions:
2  #  - lowercase term identifies a non-terminal
3  #  - uppercase term identifies a terminal, whose lexeme should be deducible from label, except:
4  #    > IDENTIFIER ::= (UNDERSCORE)* LETTER (UNDERSCORE | DOT | LETTER | DIGIT | MINUS)*
5  #    > INTEGER_LIT ::= (DIGIT)+          > FLOAT_LIT ::= (DIGIT)+ (DOT (DIGIT)+)?
6  #    > STRING_LIT ::= DQUOTE (~'"')* DQUOTE  // non-double-quote characters
7
8                script ::= MODULE IDENTIFIER SEMICOLON
9                           (import)*
10                          (function)*
11                          (tactic)*
12                          (strategy)*
13                          EOF ;
14
15               import ::= IMPORT (LIB|MODEL|OP) STRING_LIT importRenameClause? SEMICOLON ;
16   importRenameClause ::= LBRACE importRenamePhrase (COMMA importRenamePhrase)* RBRACE ;
17   importRenamePhrase ::= IDENTIFIER AS IDENTIFIER ;
18
19             function ::= DEFINE dataType IDENTIFIER ASSIGN expression SEMICOLON ;
20             operator ::= identifierPrimary ;
21
22               tactic ::= TACTIC signature LBRACE
23                            ( declaration SEMICOLON )*
24                            CONDITION LBRACE (booleanExpression SEMICOLON)* RBRACE
25                            ACTION LBRACE (statement)* RBRACE
26                            EFFECT LBRACE (booleanExpression SEMICOLON)* RBRACE
27                          RBRACE ;
28
29             strategy ::= STRATEGY IDENTIFIER
30                          LBRACKET booleanExpression RBRACKET
31                          LBRACE (function)* (strategyExpr)* RBRACE ;
32         strategyExpr ::= IDENTIFIER COLON strategyCond IMPLIES strategyOutcome ;
33         strategyCond ::= LPAREN (HASH LBRACKET strategyProbValue RBRACKET)?
34                          (booleanExpression | SUCCESS | DEFAULT) RPAREN ;
35      strategyOutcome ::= strategyClosedOutcome SEMICOLON
36                        | strategyOpenOutcome (AT LBRACKET expression RBRACKET)?
37                          LBRACE (strategyExpr)+ RBRACE ;
38 strategyClosedOutcome ::= DONE | FAIL
39                        | DO LBRACKET (IDENTIFIER | INTEGER_LIT)? RBRACKET IDENTIFIER ;
40   strategyOpenOutcome ::= IDENTIFIER LPAREN argExpressionList RPAREN
41                        | NULLTACTIC ;
42     strategyProbValue ::= FLOAT_LIT | IDENTIFIER (LBRACE IDENTIFIER RBRACE)?
43
44     argExpressionList ::= expression (COMMA expression)* ;
45 # boolean, quantified, logical, relational, and arithmetic expressions
46           expression ::=  ... ;  # see lines 82-109 in Appendix B on page 176
47 # one of the expressions is a "primaryExpression," which may be an "identifierPrimary"
48     identifierPrimary ::= IDENTIFIER (LPAREN argExpressionList RPAREN)? ;
49 # compound stmt, declaration, expression, if-stmt, for-stmt, while-stmt, or operator
50            statement ::= ... | operator ;
51
52            signature ::= IDENTIFIER
53                          LPAREN (dataType IDENTIFIER (COMMA dataType IDENTIFIER)*)? RPAREN ;
54          declaration ::= dataType IDENTIFIER (ASSIGN expression)?
55                          (COMMA IDENTIFIER (ASSIGN expression)?)* ;
```

Figure 4.1: Stitch grammar highlights (see Appendix B on page 175 for full grammar)

4.3.4 Tactic

A *tactic*, as distinguished from the *operator*, provides an abstraction that (a) packages operators into larger units of change to form primitive steps of adaptation, and (b) serves as a logical unit for specifying the cost and benefit impact of an adaptation step with respect to the quality dimensions. In Stitch, the tactic construct is characterized as follows:

- It specifies a sequence of operators;
- It is guarded with a set of conditions that determine its applicability;
- It defines a set of effects that should be observed after sequence completion;
- Its execution context consists of the architectural style of the system (for reference to the architectural types and operators) and model instance;
- During execution it accesses a read-only snapshot of the model instance, captured at the time the tactic is invoked;
- It uses only operators, and provides action primitives to the strategy (Section 4.3.5); and
- It is specified with impacts on the quality dimensions of the system (Section 4.3.6).

The tactic offers an abstract primitive for adaptation that has separate concerns from the style-defined operator to modify system elements. It provides a construct for attributing the impact of adaptation on the target system, while allowing multiple operations to be packaged into a larger, but conceptually single, step of change (e.g., setFidelity(1) for active servers).[5] As a single adaptation step, the tactic has access only to a snapshot of the architecture model at the start of execution. The snapshot is not updated during tactic execution, so conditional statements within the tactic can depend only on the initially known architectural state and not on intermediate changes in system state. This single-step semantics also means that tactics cannot invoke other tactics. Nesting tactics makes cycles possible, complicating not only the tactic's single-step semantics, but also the evaluation of its condition and effect blocks (cf. Section 8.4).

The Stitch grammar for the *tactic* is shown in Figure 4.1, lines 22–27. Before defining tactics, the script first imports the namespace of the architecture model and the architectural operators using the import statement (lines 15–17). The import statement allows renaming references with shorter identifiers. As noted in Section 4.1, the Rainbow framework binds model references to the model managed by the Model Manager, and translates the operators to system-level effectors. In the tactic, the condition block (line 24) defines the condition of applicability for the tactic. The action block (line 25) prescribes the specific operations for performing the tactic; it allows assignment statements and basic control flow statements such as if-then-else, while-loop, and for-loop. The effect block (line 26) defines the expected effect after executing the action block. Both the condition and effect blocks allow first-order predicate expressions over model properties.

An example tactic to switch servers to textual mode is illustrated in Figure 4.2. The script first imports (lines 1–3) the architectural style (T), the model instance (M),[6] and the style operators. The condition block (lines 6–8) specifies that there must exist at least one client component whose experienced-response-time property exceeds a maximum threshold. The action block (lines 9–

[5]Note that this approach leads to no loss of generality because a single operator can be wrapped as a tactic.

[6]Note that reference to the model instance is necessary for quantification over generic instances of a certain type (e.g., components of type T.ClientT), but does not impact reuse of the tactic nor the strategy.

```
1  module newssite.tactics.example;
2  import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3  import op "newssite.operator.ArchOp" { ArchOp as Sys };
4
5  tactic switchToTextualMode () {
6    condition {
7      exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
8    }
9    action {
10     svrs = { select s : T.ServerT | !s.isTextualMode };
11     for (T.ServerT s : svrs) {
12       Sys.setTextualMode(s, true);
13     }
14   }
15   effect {
16     forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
17     forall s : T.ServerT in M.components | s.isTextualMode;
18   }
19 }
```

Figure 4.2: An example *tactic* switchToTextualMode

14) finds all the servers not already in textual mode and sets them to textual mode via a model-provided operator with the signature, setTextualMode(ServerT, boolean). The effect block (lines 15–18) specifies that no Client should exhibit an above-threshold response time, and that all Servers should be in textual mode.

A tactic terminates in one of four ways. In the normal case, the condition block evaluates to true, signifying that the tactic is applicable; the action block completes, signifying no operators have failed; and the effect block evaluates to true, signifying the tactic has achieved its aim. We can identify three unsuccessful situations: (1) the condition block evaluates to false, signifying inapplicability; (2) the condition block evaluates to true, but the action block fails to complete, signifying some operator has failed; (3) the condition block evaluates to true, the action block completes, but the effect block evaluates to false, signifying that the tactic has not achieved its aim. We will see how tactic failure is handled in the strategy section below.

Cost-Benefit Attributes *Cost-benefit attributes* define the impact of a tactic on each of the quality dimensions (e.g., disruption time, content fidelity), represented as a vector of dimension-value pairs, where the value component captures delta value of cost or benefit. Thus, if a target system owner defines three quality dimensions, u_1, u_2, u_3, then the attribute vector would look like $[a_1 : \Delta v_1, a_2 : \Delta v_2, a_3 : \Delta v_3]$. For instance, tactic enlistServers for Znn.com is defined with the cost-benefit attribute vector, $[a_R : -1000, a_F : 0, a_C : +1.00]$, which specifies that the tactic is expected to reduce the average response time by 1000, to have no impact on content fidelity, and to increase cost by 1. The delta values specify no units, but they are understood to have the units of the quality dimension as defined in the utility profile. Excerpted from Section 5.1.5, the example is shown below. By defining the cost-benefit attributes for a tactic, we capture causal relationships between an adaptation step and the quality dimensions.

```
1    enlistServers:
2      uR: −1000
3      uF: 0
4      uC: +1.00
```

4.3.5 Strategy

A *strategy* encapsulates a path of adaptations where each step is the conditional execution of some tactic. Conditions along the path allow us to make the path sensitive to how effective the tactics are. Key features of the strategy are: (1) The choice of a tactic might depend on intermediate result; in some cases, an intermediate step might put the system temporarily in a state that provides fewer services or, more generally, reduces utility. (2) By introducing timing delay, we can account for asynchrony in achieving the effect of a tactic on the target system. (3) Probabilities on conditions allow us to handle inherent uncertainty about the effects of operations on the system (useful for calculating expected utility, Section 4.4.2). In Stitch, the strategy is characterized as follows:

- It is a tree of condition-action-delay decision nodes, with explicitly defined probability for conditions and delay time-window for observing tactic effects;
- It specifies conditions of applicability as style, system, or quality domain constraints;
- Its execution context consists of the system style (for the types) and the set of tactics;
- During execution it allows intermediate system observations at each decision node; and
- It enables the computation of *aggregate cost-benefit attributes* for utility-based strategy selection (Section 4.3.6).

The strategy embodies explicit decision choices and provides a packaging construct to constrain adaptation to individual domains of expertise for tractable human reasoning. Its condition-action-delay construct explicitly allows observation of the target system at decision points and provides an intuitive handle on nondeterministic outcomes. The strategy packages multiple adaptation steps as a conceptual unit of adaptation that improves the target system, even if an intermediate step temporarily reduces target-system utility. This conceptual unit is the level at which composition of adaptations is feasible for achieving multiple objectives. A strategy cannot invoke another strategy (cf. Section 8.4), as it is unclear (a) how to reconcile the conditions of applicability in the called strategy, (b) what it would mean for a strategy written for one quality to invoke a strategy intended for a different quality, and (c) how to handle recursive invocations. Furthermore, utility-based scoring of nested strategies would be greatly complicated.

We model the *strategy* syntax after Dijkstra's Guarded Commands Language [Dij75], which provides condition-action and while-loop constructs. The *strategy* grammar is shown on lines 29–42 of Figure 4.1. Before defining strategies, as before, the script first imports the namespace of the Rainbow model, tactics, and Rainbow-provided library utilities. Next, the script defines functions and then strategies. Functions (line 19) can appear at the beginning of a Stitch script or a strategy and can access the model. They compute one type of value from an expression (lines 45–46). The complete semantics and syntax of a strategy are explained through a simple strategy from the Znn.com example to reduce system response time, shown in Figure 4.3. At the top, the script imports (lines 2–5) the namespace of the architecture (T & M) and environment (E) models, the tactic defined in Figure 4.2, and library utilities (Model, for querying the model).

The strategy is first specified with its condition of applicability, usually evaluated against the model. Strategy SimpleReduceResponseTime specifies the expression, styleApplies && cViolation, as its condition of applicability (line 11). The Boolean function styleApplies (line 7) checks whether the model defines two architectural types used in the strategy. The Boolean function

```
1  module newssite.strategies.example;
2  import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3  import model "ZnnEnv.acme" { ZnnEnv as E };
4  import lib "newssite.tactics.example";
5  import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
6
7  define boolean styleApplies = Model.hasType(M, "ClientT") && ... "ServerT" ...;
8  define boolean cViolation =
9      exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
10
11 strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
12   define boolean hiLatency =
13     exists conn : T.HttpConnT in M.connectors | conn.latency > M.MAX_LATENCY;
14   define boolean hiLoad =
15     exists s : T.ServerT in M.components | s.load > M.MAX_UTIL;
16
17   t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
18     t1a: (success) -> done ;
19   }
20   t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
21     t2a: (!hiLoad) -> done ;
22     t2b: (!success) -> do [1] t1 ;
23   }
24   t3: (default) -> fail;
25 }
```

Figure 4.3: An example *strategy* SimpleReduceResponseTime

cViolation (lines 8–9) checks for the violation condition that any client is experiencing above-normal response time. If the applicability condition evaluates to false at strategy selection time, then the Adaptation Manager would not include this strategy for utility-based selection. On the other hand, if the applicability condition is satisfied and this strategy is selected, then the top-level nodes in the strategy body, labeled t1, t2, and t3 in this example, are considered.

The example strategy defines two Boolean functions, hiLatency (lines 12–13) and hiLoad (lines 14–15). In node t1 (lines 17–19), if condition hiLatency evaluates to true, then tactic switchToTextualMode is executed (line 17 after arrow). Since there is typically a delay to observe the outcome of tactic execution in the system, t1 specifies up to 1000 milliseconds (end of line 17) for observing the tactic effect. The only child node of t1 is t1a (line 18). In a strategy, the keyword **success** is a Boolean condition to indicate whether the parent-node tactic completed successfully. In the example, the **success** condition of node t1a evaluates to true if tactic switchToTextualMode completes successfully, or false if the tactic fails due to one of the three situations identified in Section 4.3.5. In the successful case, the keyword **done** is evaluated.

In a strategy, the keyword **done** signifies that the adaptation aim of the strategy has been achieved and terminates the strategy successfully, while **fail** signifies that the adaptation aim has not been achieved and aborts the strategy (a Rainbow administrator is alerted of this outcome). The keyword **default** is a Boolean condition to indicate whether *none* of the peer-node conditions in the branch level matches. If no **default** branch is defined, then one is implied with a **fail** terminator. In the example strategy on line 188, under unsuccessful cases, since no other peer nodes match, the implied **default** branch is chosen.

If condition hiLatency evaluates to false but hiLoad is true, then node t2 (lines 20–23) would be evaluated and tactic enlistServer executed to enlist an available server. Again, due to asynchrony,

t2 specifies up to 2000 ms for observing the tactic effect. In the case that hiLoad is true and tactic enlistServer does not succeed, child node t2b is chosen and the **do**-repetition is evaluated, which is semantically equivalent to repeating the subtree rooted at node t1 as a child node, *t1'*, of t2b. Evaluation proceeds with the hiLatency condition of node *t1'*: if true, then the rest of the node is evaluated as described before; if false, then the implied **default** branch aborts the strategy. The [1] (default to 3 if unspecified) indicates the number of times the **do** repetition can occur within a single evaluation of this strategy if the **do** node is revisited (not illustrated by this example).

In the example strategy, if both top-level node conditions hiLatency and hiLoad are true, then one is chosen randomly. An alternative to choose by utility would not only add computation overhead, but also render the original strategy utility score meaningless. If neither hiLatency nor hiLoad is true, then node t3 is chosen, and the strategy aborts. Note that this strategy has five termination points: nodes t1a and t2a define two success points, node t3 defines an explicit failure point, and nodes t1 and t2 have two implicit child-level failure points.

When a strategy fails, any unresolved adaptation conditions are handled subsequently through the normal cycle of triggering adaptation conditions and further strategy execution. The Adaptation Manager also has a basic learning feature: It tracks the historical failure rate of each strategy—its failure count divided by selection count—and incorporates that failure rate in the scoring process for strategy selection. The rationale for tracking just the failure rate is that, when a strategy g has been selected for execution, it is the best match for some set of conditions c. Failure of g decreases the confidence that c would indicate the applicability of g the next time around, without needing to consider the exact conditions under which g failed, so no additional statistics need to be tracked for g. Once the adaptation engineer turns on this learning feature and defines the *strategy-failure* utility profile, the framework automatically incorporates strategy failure rates as a utility component in weighted-scoring. This feature gives the adaptation engineer control over how likely a strategy is selected in the future if it has been prone to failure.

Because there is uncertainty in whether (a) a tactic achieves its intended effect and (b) a condition is observed, we estimate the likelihood of observing the branch conditions. As we shall see in Section 4.4.2, stochastic branch conditions allow an aggregate attribute vector to be computed for each strategy in utility-based selection, described next. The probabilities can be captured explicitly for each strategy node condition. The Pr{*} expression preceding each of the conditions of t1 and t2 (lines 17 and 20) denotes the estimated probability that the condition will evaluate to true among the peer-node conditions. Actual probability values are defined in a separate property file and not significant during strategy evaluation. Probabilities for peer-node conditions sum to 1; thus, in the example, t3's probability is an implied complement.

Although we have used a simplified strategy to illustrate the Stitch syntax, Stitch is sufficiently expressive for defining more elaborate strategies. For example, one could use loops to repeat a path of tactics until a condition is observed. One could also use set quantification to condition adaptations on a partial subset of model elements. A couple of more interesting examples from the Znn.com system can be found in Appendix C on page 184.

4.3.6 Strategy Selection

When faced with a system problem and multiple adaptation alternatives to choose from, the sysadmin considers business objectives and combines heuristics with past experiences to analyze

trade-offs and determine the best course. To automate this process, in Rainbow the Adaptation Manager weighs the alternative strategies against the quality dimensions by evaluating the expected aggregate utility of each applicable strategy using the specified utility profiles and preferences for the quality dimensions.

Once we have defined cost-benefit attributes for each tactic and specified utility preferences over the quality dimensions, we are only steps away from selecting a strategy (formalized in Section 4.4.2) that is most applicable to the current system conditions and that balances across the quality dimensions. The next step is to compute the expected aggregate attribute vector for each strategy. Recall that a strategy is composed of a tree of tactics, and the condition of each node has a likelihood of matching. Computing the aggregate vector consists of descending the strategy tree, unfolding **do**-repetitions as described in the *strategy* section above, and collecting the cost-benefit attribute values of each tactic. If a **do** node is revisited more than once, and no **do** counter is specified, then it is unfolded a finite number of times so that the recursion terminates. (Although the default count is 3, we could instead terminate recursion when the net impact to the path probability of unfolding another step falls below some small ϵ, e.g., $\epsilon < 0.01$.)

The probabilities defined (or implied) for the conditions allow the attribute values to be propagated up the strategy tree and eventually collected into an expected aggregate attribute vector. The vector values designate aggregate delta costs and benefits against the quality dimensions, so they are combined with architectural property values representing current system conditions, as identified by the mappings of corresponding quality dimensions (e.g., the exponential average of the client-experienced response time for the u_R dimension). Using the utility profiles, these aggregate values are converted to utility values per dimension, then a weighted sum is computed using the preference weights to yield an expected utility score for each strategy. Finally, the Adaptation Manager selects the strategy with the highest score, which is expected to achieve the highest system utility when executed successfully.

Having presented the core features of Stitch, we now explore the semantics of its constructs.

4.4 Semantics of Stitch Constructs

We define the self-adaptation semantics of Stitch in correspondence with three important aspects of the language: adaptation constructs, utility-based strategy selection, and execution of adaptation. In this section, we first present an abstract machine and mathematical process to formalize the adaptation process and establish a common mathematical basis to relate the Stitch constructs. We then describe an algorithm for computing aggregate cost-benefit attribute vectors to facilitate utility-based strategy selection. Finally, we present the operational semantics for evaluating a Stitch script to execute an adaptation during the adaptation process.

4.4.1 Model of Adaptation

We now present an abstract machine and mathematical process to formalize the Rainbow adaptation process and provide semantics for the Stitch concepts: *adaptation conditions*, *tactic*, and *strategy*. As we will see, the core constructs of Stitch, summarized in Table 4.1 and defined

by the grammar as in Figure 4.1, correspond to elements of a Markov Decision Process (MDP) (except for one minor mismatch discussed in Section 8.4).

A Markov process is a mathematical model that is useful in the study of complex systems, using the basic concepts of *state* of a system and state *transition* [How60]. An MDP is a mathematical framework built upon the concept of a Markov process to model problems in which outcomes are partly under the control of a decision maker and partly nondeterministic. Formally, an MDP is a discrete-time stochastic control process defined by a four-tuple,

$$\mathbf{M} = (S, A, P_{\cdot}(\cdot,\cdot), R(\cdot))$$

where

- S is a countable set of possible states, with an initial state S_0
- A is a countable set of actions
- $P_{\cdot}(\cdot,\cdot)$ is a transition probability matrix (by definition, the rows sum to 1), and
- $R(\cdot)$ is an immediate reward function with range $[0, 1]$

The state of the system changes over time due, in part, to the action choice of the decision maker. In each state, $s \in S$, there are a number of actions, $a \in A$, from which the decision maker may choose. The destination state, $s' \in S$, is determined according to the transition probability matrix $P_{\cdot}(\cdot,\cdot)$. Specifically, $P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ where t is the current time. That is, $P_a(s, s')$ is the probability of going into state s' at time $t+1$ when the decision maker chooses action a while in state s at time t. When the decision maker is in state s, an immediate reward equal to $R(s)$ is earned. Earned rewards accrue over time.

The goal of the decision maker is to accumulate the maximum reward possible over time, typically with a discount rate of γ on future rewards, $\sum_{t=0}^{\infty} \gamma^t R(s_t)$, where $0 < \gamma \leq 1$ (usually a few percent under 1). This is accomplished by solving for the optimal policy Π, which specifies the best action to choose in each state.

To formalize Stitch, we start with the *managed system* and represent its states as conjunctions of its architecture-model predicates. Formally, let AP be a set of atomic propositions—i.e., Boolean expressions over variables, constants, and predicate symbols—which compose the set of architectural predicates. We define L as an interpretation function that maps a set of APs from the architecture model to a state in the MDP, $L : 2^{AP} \rightarrow S$. For example, if the architecture model indicates an instance s: ServerT with a load property, s.load, that is below a threshold value $Load_{max}$, then the following conjunction describes this example state, $s_{eg} \in S$:

$$s_{eg} = L\left(hasInstance(s) \wedge declaresType(s, ServerT) \wedge \neg s.load \geq Load_{max}\right)$$

We define a *trap state* to capture error states. Furthermore, we interpret the complement of the MDP discount factor, $1 - \gamma$, as the probability of an error occurring that is not modeled in any of the states. For the MDP to be useful, $1 - \gamma$ is usually a low, single-digit percentage value; otherwise, the MDP reaches negligible probability after a few iterations.

Let T be the set of tactics. A tactic $t \in T$ is defined as a sequence of operators (whose semantics are unspecified in our model). Given a set, $OPERATOR$,[7] define $T : \mathbf{seq}\, OPERATOR$,

[7]We will represent unspecified types (without further structure) in all caps, similar to Z *basic types*.

and define the *null tactic*, $t_{NULL} \in T$, as the null sequence, $t_{NULL} = \langle\,\rangle$. t_{NULL} is unique because

$$\exists t : T \bullet t = t_{NULL} \wedge \forall t' : T \mid t' = t_{NULL} \bullet t = t'$$

Tactics are treated as atomic adaptation primitives each corresponding to an action, $a \in A$, which transitions the system nondeterministically from a state s to a destination state s', i.e., $T \subseteq A$.

Let G be the set of strategies. A strategy $g \in G$ is defined as a tree over a set of vertices V and a set of edges $E : V \times V$. The vertices correspond to the tactics, $V \subseteq T$. To simplify predicates that characterize the strategy, we first define a few convenience functions:

$$root : G \rightarrow V$$

$$children : V \rightarrow \mathbb{R}V$$

$$edge : V \times V \rightarrow E$$

$$pred : E \rightarrow V$$

$$succ : E \rightarrow V$$

$$cond : E \rightarrow \mathbb{P}AP$$

$$prob : E \rightarrow [0, 1]$$

The function $root\,(g)$ gives the tree root of strategy g and $children\,(v)$ gives the child vertices of vertex v; $edge\,(v_1, v_2)$ gives the edge that connects the two vertices v_1, v_2; $pred\,(e)$ gives the predecessor vertex and $succ\,(e)$, the successor vertex, of e; $cond\,(e)$ defines the matching condition for edge e as a predicate of atomic propositions; and $prob\,(e)$ defines the likelihood of a transition across edge e.

To allow the aggregate expected value to be computed meaningfully over a strategy tree (see Section 4.4.2 below), the probabilities of all the branches from a vertex must sum to 1:

$$\forall v \in V \bullet sum\,(\{e \in E \mid v = pred\,(e) \bullet prob\,(e)\}) = 1$$

To handle the situation where no branch from a vertex has a matching condition, we require every source vertex to define a default branch that applies when no other conditions match:

$$\forall v \in V \bullet \exists e \in E \mid v = pred\,(e) \bullet cond\,(e) = AP - \bigcup_{e' \in E, e' \neq e} cond\,(e') \wedge succ\,(e) = FAIL$$

In relation to the MDP model, $\mathbf{M}$, the condition on every branch maps to corresponding state propositions, and the estimated probability of a transition that results from a tactic maps to the transition probability of a corresponding action:

$$\begin{aligned} \forall e \in E \bullet \exists a_1, a_2 \in A \mid a_1 = pred\,(e) \wedge a_2 = succ\,(e) \bullet \\ \exists s_1, s_2, s_3 \in S \mid P_{a_1}\,(s_1, s_2) > 0 \wedge P_{a_2}\,(s_2, s_3) > 0 \bullet \\ P_{a_1}\,(s_1, s_2) = prob\,(e) \wedge L(cond\,(e)) = s_2 \end{aligned}$$

In essence, strategy g maps into a subgraph of $\mathbf{M}$, where the branch condition of a node t_0 in g corresponds to a state s_{t0} in $\mathbf{M}$, the node t_0 corresponds to a transition action a_{t0} from s_{t0}, the branches from t_0 correspond to the nondeterministic target states from a_{t0}, and so on. The probabilities on the strategy branches correspond to the MDP transition probabilities. The tactic observation delay time in the strategy is captured as an action effect of the MDP transition.

4.4.2 Utility-Based Strategy Selection

We now present the semantics for the Stitch concepts *quality dimensions*, *cost-benefit attributes*, *utility preferences*, and *strategy selection*, and describe an algorithm to compute the expected aggregate cost-benefit attribute vector, $EAAV$, for strategy selection.

Quality dimensions enable quantitatively capturing goals for achieving certain runtime quality attributes, where each attribute has a name and a range of values. An attribute might be a cost, such as resource consumed, or a benefit, such as reduction in response time, but in computing aggregate values, we accumulate units of each in the same manner, so we need not make a formal distinction. We uniformly accumulate values by the sum operator $+$ regardless of attribute type because, at the strategy level, tactics are evaluated sequentially. Thus, we formalize a quality attribute simply as an identifying label (e.g., "disruption") and a value range, where the unspecified range type might be real, enumerations, or even a dimensional unit, but it must be cardinal:

$$QT \triangleq (LABEL, RANGE)$$

For each target system, we define a set of *quality attributes*, A, with q elements of QT:

$$A \equiv \{a_i, \cdots, a_q \in QT\}$$

The precise list differs by system, but we show one example below:

$$A_{eg} \equiv \{a_1, a_2, a_3, a_4\} = \{(\text{budget}, \$), (\text{disruption}, \{1,2,3,4,5\}), \\ (\text{response\_time}, \mathbb{N}\ \text{ms}), (\text{content\_fidelity}, \{1,2,3,4,5\})\}$$

The set A forms an q-dimensional space of attribute values, SAV, for the target system, each dimension having the range defined by a_i:

$$SAV \triangleq RANGE_1 \times \cdots \times RANGE_q$$

A point in this space is represented by a vector $(v_1, \cdots, v_q)$, subscripted by dimension $i \in \mathbb{N}[1, q]$, where each value v_i falls within the range defined by a_i.

For each tactic, there is a *cost-benefit attribute vector* that describes the expected cost incurred and benefit delivered by a tactic when it completes. The value elements in this vector represent delta cost or delta benefit values for corresponding attributes; hence, we define a delta space,

$$dSAV \triangleq dRange(1) \times \cdots \times dRange(q)$$

where the values of each dimension i range from the negative to the positive of the maximum magnitude of differences between any values of $RANGE_i$ in SAV:

$$dMax(R : \mathbb{P}RANGE) \equiv max\{v_1, v_2 \in R \bullet |v_1 - v_2|\}$$

$$dRange(i \in \mathbb{N}[1, q]) \equiv [-dmax(RANGE_i), +dmax(RANGE_i)]$$

We define a function, $cbav : T \rightarrow dSAV$, to get the cost-benefit attribute vector of tactic t.

A utility curve captures how happy the target system owner would be with particular values of an attribute. We define a set of utility functions where each element is a function that maps the value of the corresponding attribute $a_i \in A$ to a utility value in the range $\mathbb{R}[0, 1]$:

$$u_i : RANGE_i \rightarrow \mathbb{R}[0, 1]$$

The overall utility function, U, computes the scalar utility of a point in the attribute vector space, in effect accounting for relative priorities between the attributes. To compute U, we define the relative priorities between the attributes by a set of weights, W, which must sum to 1:

$$W \equiv \{w_1, \cdots, w_q \in \mathbb{R}\,[0,1]\}\,, \text{ where } \sum_{i=1}^{q} w_i = 1$$

$$U : SAV \rightarrow \mathbb{R}\,[0,1] = \sum_{i=1}^{q} w_i u_i$$

Together, the tuple (A, W, U) defines the utility profile and preferences for the target system to enable utility-based strategy selection. The utility function U corresponds to the expected immediate reward function $R(\cdot)$ in MDP: it computes the utility score of an architectural state.

Using the branch probabilities and the attribute vector of each tactic in a strategy tree, we can compute the *expected aggregate attribute vector*, $EAAV : G \rightarrow SAV$, over the strategy at its root node. Given any tree node x, with children nodes $children\,(x)$ and branch probabilities defined by $p\,(x,c) \equiv prob\,(edge\,(x,c))$, the aggregate attribute vector function, $AggAV$, recursively computes the sum of (a) the cost-benefit delta values of the current node and (b) the aggregate attribute vector of each child node weighted by its branch probability; at a leaf node, the second term yields zero:

$$AggAV\,(x) = cbav\,(x) + \begin{cases} 0 & \text{if } children\,(x) = \oslash \\ \sum_{c \in children(x)} p\,(x,c)\, AggAV\,(c) & \text{otherwise} \end{cases}$$

Then, $EAAV$ is the vector sum of the aggregate attribute vector at the root of a given strategy, with the attribute vector of the current system state, $sysAV \in SAV$:

$$EAAV\,(g) = sysAV + AggAV\,(root\,(g))$$

Finally, we select from the set of applicable strategies the one that yields the maximum utility,

$$\max_{g \in G} \{U\,(EAAV\,(g))\}$$

The strategy selection algorithm directly implements the mathematics described above, using dynamic programming for the recursive aggregate attribute vector function. The complexity of the algorithm depends on the number of quality dimensions, $q = |A|$, the number of nodes per strategy tree, $n = |g \in G|$, and the number of strategies, $s = |G|$.

For function $AggAV$, in the worst-case, at each node and for each dimension, three computation operations are required for attribute value lookup, addition, and multiplication ($3qn$). For function $EAAV$, one addition is required per quality dimension (q). For function U, in the worst-case, six arithmetic operations are required per dimension, u_i, for linear extrapolation using the slope-intercept formula: $y = \frac{y_2 - y_1}{x_2 - x_1}\,(x - x_2) + y_2$; a multiplication is required per weight, w_i; and a total of $q - 1$ additions are required to compute the sum ($6q + q + q - 1$).

In sum, the algorithm before the selection step requires $(3n + 9)\,q - 1$ operations per strategy. Finally, the $\max$ computation requires a total of $s - 1$ comparisons, so the overall strategy selection algorithm requires $[s\,((3n + 9)\,q - 1) + s - 1]$ operations, which reduces to $(3n + 9)\,qs - 1$.

For large numbers of dimensions, nodes per strategy, and strategies, the complexity of the utility-based strategy selection algorithm is $O(nqs)$.

Utility theory enables fair comparison of strategies that adapt for different quality dimensions. It allows us to dynamically compute trade-offs between possibly conflicting interests, provided we can elicit and capture business utility preferences and estimate strategy branch probabilities (see Section 9.2.2). By allowing strategies of that address particular quality dimensions to be chosen to adapt the target system, a utility theoretic approach allows composing adaptations across multiple dimensions to accomplish multiple, possibly conflicting objectives.

4.4.3 *Adaptation Execution*

Here we present a big-step operational semantics[8] for evaluating a Stitch script, particularly the strategy and tactic. We first define the metavariables, which identify different types of terms. We then present Stitch's abstract syntax and the evaluation rules. We conclude this section with flowcharts to illustrate strategy and tactic execution in the context of the adaptation process.

In Stitch, the execution context depends on the architecture and environment models, which are available within a script once imported by the script. For simplicity, we do not specify model structures but treat the models as a blackbox, accessible via M. Assume that, during Stitch script parsing, the parser creates a function table (FT) for the **define** statements in the script, a method table (MT) for the invoked methods, and a strategy node table (NT) for the strategy nodes of each strategy. The function table allows looking up the defined expression of a function; the method table, the body of a method. The node table allows looking up a strategy node by the node label, and ensures that label l is unique within the scope of a strategy.

v	$\in$	$\mathbf{Values}$	values: true, false, $\mathbb{N}$, $\mathbb{R}$, $\mathbf{String}$
nv	$\in$	$\mathbf{Num}$	number literals
S, AS, SAS	$\in$	$\mathbf{Stmt}$	statements
x, y, arg	$\in$	$\mathbf{Var}$	program variables; rv, rop, rtt, rsg: return variables
a	$\in$	$\mathbf{AExp}$	arithmetic expressions
P	$\in$	$\mathbf{BExp}$	Boolean predicates
s	$\in$	$\mathbf{SExp}$	set expressions
e	$\in$	$\mathbf{Exp}$	expressions ($\mathbf{Exp} = \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{SExp}$)
m	$\in$	$\mathbf{Method}$	invokable methods; rv: method return value
f	$\in$	$\mathbf{Function}$	defined functions; f_a, f_b, f_s: corresponding value types
$archOp$	$\in$	$\mathbf{ArchOp}$	architectural operators; rop: Boolean return state
l	$\in$	$\mathbf{Label}$	uninterpreted label identifiers
T	$\in$	$\mathbf{Tactic}$	tactics; T_{NULL}: null action; rtt: Boolean return state
G	$\in$	$\mathbf{Strategy}$	strategies; rsg: Boolean return state
GC	$\in$	$\mathbf{Cond}$	strategy conditions ($\mathbf{Cond} = \mathbf{BExp} \cup \{\text{default}\}$)
GA	$\in$	$\mathbf{Action}$	strategy actions
GN	$\in$	$\mathbf{Label} \times \mathbf{Cond} \times \mathbf{Action}$	strategy nodes
E, M	$\in$	$\mathbf{Var} \rightarrow \mathbf{Value}$	execution context for vars and arch and envt models

[8] Notational conventions: $metavariable$, $\mathbf{MetaSet}$, keyword, *programvalue*, mathvalue

MT	$\in$	$\mathbf{Method} \rightarrow \mathbf{seqVar} \times \mathbf{Stmt}$	method table
FT	$\in$	$\mathbf{Function} \rightarrow \mathbf{seqVar} \times \mathbf{Exp}$	function table
NT	$\in$	$\mathbf{Label} \rightarrow \mathbf{Cond} \times \mathbf{Action}$	strategy node table

The abstract syntax of Stitch appears below. Prefix increment and decrement expressions are rewritten as arithmetic assignment statements (e.g., $x := a + 1$). Relational operator $archQry$ represents typical queries on the architecture model, such as isConnected() or declaresType().

E	::=	$\oslash$ (empty context)
	|	$E, x \mapsto v$ (variable-value binding)
M	::=	$\oslash$
	|	$M, x \mapsto v$
G	::=	strategy $P \langle GN \rangle$
GN	::=	$l : GC\ GA$
	|	$GN_1; GN_2$
GC	::=	P_G
	|	default
GA	::=	$T \langle GN \rangle$
	|	do $a\ l$
	|	done
	|	fail
T	::=	tactic *SAS* cond P act S eff P
	|	T_{NULL}
S	::=	$skip$
	|	$archOp$
	|	AS
	|	$S_1; S_2$
	|	if P then S_1 else S_2
	|	while P do S
AS	::=	$x := a$
	|	$x := P$
	|	$x := s$
	|	$x := m$
SAS	::=	AS
	|	$SAS_1; SAS_2$
$archOp$	::=	$skip$
	|	$error$
v	::=	nv
	|	*true*
	|	*false*

	|	$\{v_1, \cdots, v_n\}$
	|	*errval*
a	::=	x
	|	f_a
	|	nv
	|	$a_1\ op_a\ a_2$
P	::=	x
	|	f_b
	|	*true*
	|	*false*
	|	$not\ P$
	|	$P_1\ op_b\ P_2$
	|	$a_1\ op_r\ a_2$
	|	forall x in $s \vert P$
	|	exists x in $s \vert P$
P_G	::=	P
	|	success
	|	$not\ P_G$
	|	$P_G\ op_b\ P$
s	::=	x
	|	f_s
	|	$\{\}$
	|	$\{v_1, \cdots, v_n\}$
	|	$\{x$ in $s \vert P\}$
e	::=	$a \mid P \mid s$
m	::=	*method* (arg)
f	::=	*function* (arg)
arg	::=	x
	|	arg_1, arg_2
op_a	::=	+ | - | * | / | %
op_b	::=	&& | || | -> | <->
op_r	::=	<= | < | == | != | > | >= | $archQry$

We now present the evaluation rules, with computation followed by reduction rules. In all cases, the complete context consists of $M_{test}; M_{op}; E$, where the two Ms are the model context and E is the variable context. The first M represents the model state against which $archQry$s are evaluated, while the second M reflects the updated model state (e.g., after an $archOp$ execution). In most cases, the two Ms reflect the same model state; however, in rules pertaining to the architectural operator and tactic, M_{test} and M_{op} differ to model the semantics that tactics do not *see* changes to the architecture model. To reduce visual clutter, the first M is often omitted; wherever only E appears, $M; M$ are implied in order, and they can be in an arbitrary state.

Note that rule reduce-sequenceNormal indicates the possibility of M_{op} changing arbitrarily between statements. In addition, two meta-functions are used in two rules: *remvElem* removes an element from a given set and returns the removed element; *random* randomly returns one of the supplied values.

$\boxed{\overline{E \vdash v \Downarrow \mathsf{value}}}$ Expression Evaluation

$$\frac{}{v \Downarrow v} \quad \text{(eval-val)}$$

$$\frac{E\,[x] = v}{E \vdash x \Downarrow v} \quad \text{(eval-var)}$$

$$\frac{E \vdash P \Downarrow x \quad E\,[x] = v}{E \vdash P \Downarrow v} \quad \text{(eval-bool)}$$

$$\frac{E \vdash a \Downarrow x \quad E\,[x] = v}{E \vdash a \Downarrow v} \quad \text{(eval-arith)}$$

$$\frac{E \vdash a \Downarrow v \quad E \vdash a' \Downarrow v'}{E \vdash a\, op_a\, a' \Downarrow v\, \mathsf{op_a}\, v'} \quad \text{(eval-oparith)}$$

$$\frac{}{E \vdash \mathit{true} \Downarrow \mathsf{true}} \quad \text{(eval-true)}$$

$$\frac{}{E \vdash \mathit{false} \Downarrow \mathsf{false}} \quad \text{(eval-false)}$$

$$\frac{E \vdash P \Downarrow v}{E \vdash not\, P \Downarrow \mathsf{not}\, v} \quad \text{(eval-not)}$$

$$\frac{E \vdash P \Downarrow v \quad E \vdash P' \Downarrow v'}{E \vdash P\, op_b\, P' \Downarrow v\, \mathsf{op_b}\, v'} \quad \text{(eval-opbool)}$$

$$\frac{E \vdash a \Downarrow v \quad E \vdash a' \Downarrow v'}{E \vdash a\, op_r\, a' \Downarrow v\, \mathsf{op_r}\, v'} \quad \text{(eval-oprel)}$$

$$\frac{\forall v \in s\ E \vdash [v/x]\, P \Downarrow \mathit{true}}{E \vdash \text{forall}\, x\, \text{in}\, s | P \Downarrow \mathit{true}} \quad \text{(eval-foralltrue)}$$

$$\frac{\exists v \in s\ E \vdash [v/x]\, P \Downarrow \mathit{false}}{E \vdash \text{forall}\, x\, \text{in}\, s | P \Downarrow \mathit{false}} \quad \text{(eval-forallfalse)}$$

$$\frac{\exists v \in s\ E \vdash [v/x]\, P \Downarrow \mathit{true}}{E \vdash \text{exists}\, x\, \text{in}\, s | P \Downarrow \mathit{true}} \quad \text{(eval-existstrue)}$$

$$\frac{\forall v \in s\ E \vdash [v/x]\, P \Downarrow \mathit{false}}{E \vdash \text{exists}\, x\, \text{in}\, s | P \Downarrow \mathit{false}} \quad \text{(eval-existsfalse)}$$

$\boxed{\overline{E \vdash archOp \Downarrow E}}$ Skip and Architectural Operator Reduction

$$\frac{}{E \vdash skip \Downarrow E} \quad \text{(reduce-skip)}$$

$$\frac{}{E \vdash error \Downarrow E, rop \mapsto \mathit{errval}} \quad \text{(reduce-error)}$$

$$\frac{}{M; M; E \vdash archOp \Downarrow M; M'; E} \quad \text{(reduce-archOpSuccess)}$$

$$\frac{}{M; M; E \vdash archOp \Downarrow M; M'; E, rop \mapsto \mathit{errval}} \quad \text{(reduce-archOpFail)}$$

$\boxed{\dfrac{}{E \vdash x := sub\,(\bar{v}) \Downarrow E, x \mapsto v}}$ Method and Function Reduction

$$\frac{E \vdash MT\,[m] = (\bar{x}, S) \quad [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n] \vdash S \Downarrow E'}{E \vdash x := \mathit{method}\,(\bar{v}) \Downarrow E, x \mapsto E'\,[rv]} \quad \text{(reduce-methodinvoke)}$$

$$\frac{E \vdash FT\,[f] = (\bar{x}, e) \quad [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n] \vdash e \Downarrow v}{E \vdash x := \mathit{function}\,(\bar{v}) \Downarrow E, x \mapsto v} \quad \text{(reduce-functioncall)}$$

$\boxed{\dfrac{E \vdash a \Downarrow v}{E \vdash S \Downarrow E'}}$ Statement Reduction

$$\frac{E \vdash a \Downarrow v}{E \vdash x := a \Downarrow E, x \mapsto v} \quad \text{(reduce-assign)}$$

$$\frac{M; E \vdash S_1 \Downarrow M'; E' \quad M''; E' \vdash S_2 \Downarrow M'''; E''}{M; E \vdash S_1; S_2 \Downarrow M'''; E''} \quad \text{(reduce-sequenceNormal)}$$

$$\frac{M; E \vdash S_1 \Downarrow M'; E' \quad E'\,[rop] = errval}{M; E \vdash S_1; S_2 \Downarrow M'; E'} \quad \text{(reduce-sequenceError)}$$

$$\frac{E \vdash P \Downarrow \mathsf{true} \quad M; E \vdash S_1 \Downarrow M'; E'}{M; E \vdash \text{if } P \text{ then } S_1 \text{ else } S_2 \Downarrow M'; E'} \quad \text{(reduce-iftrue)}$$

$$\frac{E \vdash P \Downarrow \mathsf{false} \quad M; E \vdash S_2 \Downarrow M'; E'}{M; E \vdash \text{if } P \text{ then } S_1 \text{ else } S_2 \Downarrow M'; E'} \quad \text{(reduce-iffalse)}$$

$$\frac{M; E \vdash \text{if } P \text{ then } S_1 \text{ else } skip \Downarrow M'; E'}{M; E \vdash \text{if } P \text{ then } S_1 \Downarrow M'; E'} \quad \text{(reduce-ifthen)}$$

$$\frac{E \vdash P \Downarrow \mathsf{true} \quad M; E \vdash S; \text{while } P \text{ do } S \Downarrow M'; E'}{M; E \vdash \text{while } P \text{ do } S \Downarrow M'; E'} \quad \text{(reduce-whiletrue)}$$

$$\frac{E \vdash P \Downarrow \mathsf{false}}{E \vdash \text{while } P \text{ do } S \Downarrow E} \quad \text{(reduce-whilefalse)}$$

$$\frac{M; E \vdash S_i;\ \text{while } P \text{ do } (S; S_f) \Downarrow M'; E'}{M; E \vdash \text{for } (S_i; P; S_f) \text{ do } S \Downarrow M'; E'} \quad \text{(reduce-forloop)}$$

$$\frac{M; E \vdash x := \mathit{remvElem}\,(s)\ ;\ \text{while } s \text{!= } \{\}\ \text{do } (S;\ x := \mathit{remvElem}\,(s)) \Downarrow M'; E'}{M; E \vdash \text{for } (x : s) \text{ do } S \Downarrow M'; E'} \quad \text{(reduce-foreach)}$$

Because the state of the target system, as reflected in the architecture and environment models, can change arbitrarily, we reflect that change in M_{op} after evaluating a statement in rules reduce-tactic* and a tactic in reduce-strategyTactic. Additionally, in rule reduce-strategyTactic, notice that M_{test} "synchronizes" with M_{op} after evaluating a tactic and before the next level of strategy nodes, and the value of the context variable rop is "reset" before proceeding with the next level of strategy nodes so that the next tactic invocation does not simply fail. To capture

the semantics that, during tactic evaluation, statements do not see changes to the model, M_{test} remains unchanged. After tactic evaluation, E is only modified by the tactic return state indicating whether tactic execution succeeded or failed. In rules reduce-strategyDo*, $cnt_{l_{do}}$ is a counter variable unique to each *do* node to track the number of times the *do* node has been visited.

$$\boxed{\frac{}{E \vdash T \Downarrow E, rtt \mapsto v}}$$ Tactic Reduction

$$\frac{}{E \vdash T_{NULL} \Downarrow E, rtt \mapsto \textit{true}}$$ (reduce-nullTactic)

$$\frac{E \vdash \textit{SAS} \Downarrow E' \quad E' \vdash P_1 \Downarrow \mathsf{true} \quad M; M; E' \vdash S \Downarrow M'; E'' \quad M'; E'' \vdash P_2 \Downarrow \mathsf{true}}{M; M; E \vdash rtt := \text{tactic } \textit{SAS} \text{ cond } P_1 \text{ act } S \text{ eff } P_2 \Downarrow M; M'; E, rtt \mapsto \textit{true}}$$ (reduce-tacticSuccess)

$$\frac{E \vdash \textit{SAS} \Downarrow E' \quad E' \vdash P_1 \Downarrow \mathsf{false}}{E \vdash rtt := \text{tactic } \textit{SAS} \text{ cond } P_1 \text{ act } S \text{ eff } P_2 \Downarrow E, rtt \mapsto \textit{false}}$$ (reduce-tactic-CondFail)

$$\frac{E \vdash \textit{SAS} \Downarrow E' \quad E' \vdash P_1 \Downarrow \mathsf{true} \quad M; M; E' \vdash S \Downarrow M'; E'' \quad E''[rop] = errval}{M; M; E \vdash rtt := \text{tactic } \textit{SAS} \text{ cond } P_1 \text{ act } S \text{ eff } P_2 \Downarrow M; M'; E, rtt \mapsto \textit{false}}$$ (reduce-tactic-ActionFail)

$$\frac{E \vdash \textit{SAS} \Downarrow E' \quad E' \vdash P_1 \Downarrow \mathsf{true} \quad M; M; E' \vdash S \Downarrow M'; E'' \quad M'; E'' \vdash P_2 \Downarrow \mathsf{false}}{M; M; E \vdash rtt := \text{tactic } \textit{SAS} \text{ cond } P_1 \text{ act } S \text{ eff } P_2 \Downarrow M; M'; E, rtt \mapsto \textit{false}}$$ (reduce-tactic-EffectFail)

$$\boxed{\frac{E \vdash P \Downarrow v}{E \vdash G \Downarrow E, x \mapsto v}}$$ Strategy Reduction

$$\frac{E \vdash P \Downarrow \mathsf{false}}{E \vdash \text{strategy } P \langle GN \rangle \Downarrow E, rsg \mapsto \textit{false}}$$ (reduce-strategy-NotAppl)

$$\frac{E \vdash P \Downarrow \mathsf{true} \quad M; M; E \vdash \langle GN_1; \cdots; GN_n \rangle \Downarrow M'; M'; E' \quad E'[rsg] = v}{M; M; E \vdash x := \text{strategy } P \langle GN \rangle \Downarrow M'; M'; E, x \mapsto E'[rsg]}$$ (reduce-strategyComplete)

$$\boxed{\frac{E \vdash GC \Downarrow v}{E \vdash GN \Downarrow E'}}$$ Strategy Condition Reduction

$$\frac{E[rtt] = v}{E \vdash \text{success} \Downarrow v}$$ (reduce-strategy-CondSuccess)

$$\frac{E \vdash GC_i \Downarrow \mathsf{true} \quad E \vdash GA_i \Downarrow E'}{E \vdash \langle l_1 : GC_1\, GA_1; \cdots; l_n : GC_n\, GA_n \rangle \Downarrow E'}$$ (reduce-strategy-NodeOneMatch)

$$\frac{E \vdash GC_1 \Downarrow \mathsf{false} \cdots GC_{n-1} \Downarrow \mathsf{false} \quad GC_n = \text{default} \quad E \vdash GA_n \Downarrow E'}{E \vdash \langle l_1 : GC_1\, GA_1; \cdots; l_n : GC_n\, GA_n \rangle \Downarrow E'}$$ (reduce-strategy-NodeDefaultMatch)

$$\frac{E \vdash GC_1 \Downarrow \mathsf{false} \cdots GC_n \Downarrow \mathsf{false}}{E \vdash \langle l_1 : GC_1\, GA_1; \cdots; l_n : GC_n\, GA_n \rangle \Downarrow E, rsg \mapsto \textit{false}}$$ (reduce-strategy-NodeNoMatch)

$$\boxed{\frac{}{E \vdash GA \Downarrow E'}}$$ Strategy Action Reduction

$$\frac{M; M; E \vdash T \Downarrow M; M'; E' \quad M'; M'; E', rop \mapsto \oslash \vdash \langle GN_1; \cdots; GN_n \rangle \Downarrow M''; M''; E''}{M; M; E \vdash T\ \langle GN \rangle \Downarrow M''; M''; E''} \quad \text{(reduce-strategyTactic)}$$

$$\frac{cnt_{l_{do}} \notin \mathbf{dom} E \quad NT\,[l_{tgt}] = (GC, GA) \quad E \vdash GC \Downarrow \mathsf{true} \quad E \vdash a \Downarrow nv \quad E, cnt_{l_{do}} \mapsto nv \vdash GA \Downarrow E'}{E \vdash \mathrm{do}_{l_{do}}\, a\, l_{tgt} \Downarrow E'} \quad \text{(reduce-strategyDo1stOk)}$$

$$\frac{cnt_{l_{do}} \notin \mathbf{dom} E \quad NT\,[l_{tgt}] = (GC, GA) \quad E \vdash GC \Downarrow \mathsf{false}}{E \vdash \mathrm{do}_{l_{do}}\, a\, l_{tgt} \Downarrow E, rsg \mapsto \mathit{false}} \quad \text{(reduce-strategyDo1stFail)}$$

$$\frac{E\,[cnt_{l_{do}}] = nv \quad nv \neq 0 \quad NT\,[l_{tgt}] = (GC, GA) \quad E \vdash GC \Downarrow \mathsf{true} \quad E, cnt_{l_{do}} \mapsto (nv - 1) \vdash GA \Downarrow E'}{E \vdash \mathrm{do}_{l_{do}}\, a\, l_{tgt} \Downarrow E'} \quad \text{(reduce-strategyDoOk)}$$

$$\frac{E\,[cnt_{l_{do}}] = nv \quad nv \neq 0 \quad NT\,[l_{tgt}] = (GC, GA) \quad E \vdash GC \Downarrow \mathsf{false}}{E \vdash \mathrm{do}_{l_{do}}\, a\, l_{tgt} \Downarrow E, rsg \mapsto \mathit{false}} \quad \text{(reduce-strategyDoFail)}$$

$$\frac{E\,[cnt_{l_{do}}] = 0}{E \vdash \mathrm{do}_{l_{do}}\, a\, l_{tgt} \Downarrow E, rsg \mapsto \mathit{false}} \quad \text{(reduce-strategyDoEnd)}$$

$$\frac{}{E \vdash \mathrm{done} \Downarrow E, rsg \mapsto \mathit{true}} \quad \text{(reduce-strategyDone)}$$

$$\frac{}{E \vdash \mathrm{fail} \Downarrow E, rsg \mapsto \mathit{false}} \quad \text{(reduce-strategyFailed)}$$

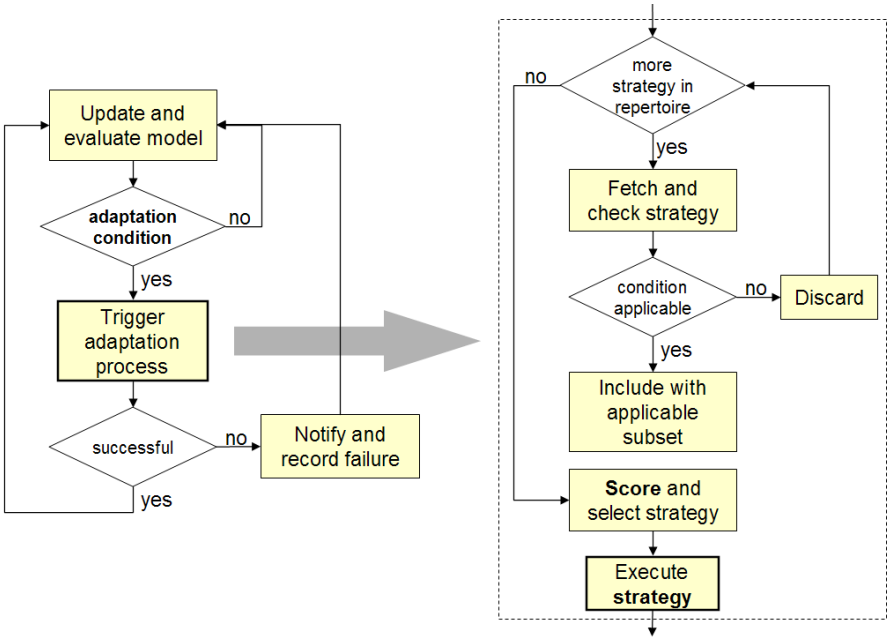


Figure 4.4: Flowchart of the Rainbow adaptation process; Figure 4.5 refines *Execute strategy*

To illustrate the execution flow of the adaptation process, Figure 4.4 shows a flowchart of the abstract Rainbow adaptation cycle on the left-hand half and the Adaptation Manager's adaptation process on the right-hand half. The *Score and select strategy* process uses the algorithm described in Section 4.4.2. The flow of the *Execute strategy* process is expanded in Figure 4.5.

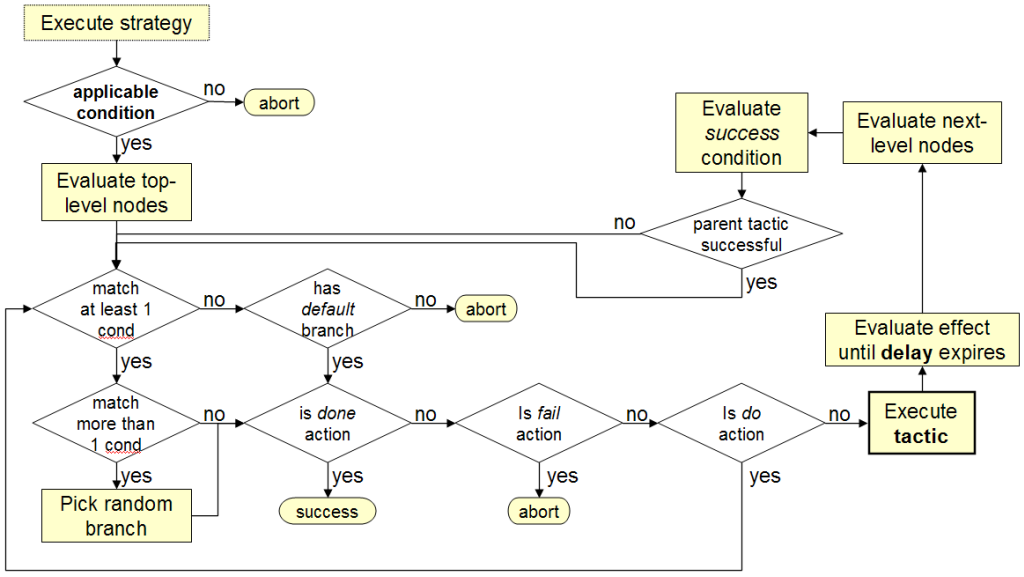


Figure 4.5: Flowchart of strategy execution, Figure 4.6 refines *Execute tactic*

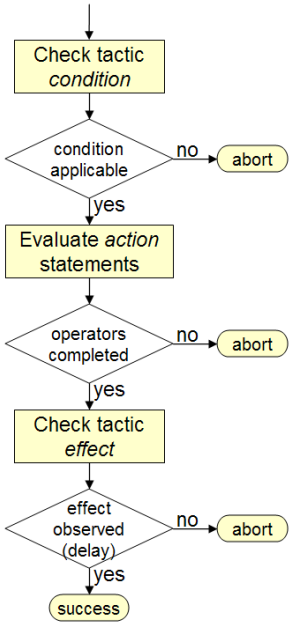


Figure 4.6: Flowchart of tactic execution

In Figure 4.5, the *abort* and *success* terminators conclude the *Execute strategy* process in Figure 4.4, which is followed by the *successful?* decision diamond: an *abort* terminator results

in a *no* decision, while a *success* results in a *yes*. Finally, the flow of the *Execute tactic* process in Figure 4.5 is expanded in Figure 4.6, with its *abort* and *success* terminators having the same effect on the *parent tactic successful?* decision diamond in Figure 4.5.

4.5 Stitch Illustration Using Znn.com

To bring together the concepts, we now illustrate the features of Stitch using the Znn.com example system. We start with the three high-level, potentially competing, objectives and specify a set of utility functions and preferences for those. We illustrate the definition of adaptation tactics with their attribute vectors and demonstrate strategy selection using the defined utility preferences.

The stakeholders in the Z.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 a tactic, we also need to consider a fourth dimension, disruption, which should be minimized. We use an ordinal scale of 1 to 5 to express the degree of disruption. Note that these four quality dimensions provide the corresponding cost-benefit attributes necessary for strategy selection.

Given our understanding of the quality dimensions, we can specify discrete utility functions for these four dimensions and complete the utility profiles (comments elided). To determine the utility preferences, assume that Znn.com considers response time the most important, followed by budget, then content quality, and finally disruption. This might yield a linear set of relative weights shown in Table 4.7.

Table 4.7: Znn.com utility profiles and preferences

u_R	Avg Response Time	ClientT.experRespTime	$\langle(\mathsf{low}, 1), (\mathsf{medium}, 0.5), (\mathsf{high}, 0)\rangle$	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

As described in the scenario (Section 3.2), four adaptations are possible and can be fulfilled with three tactics. A switchToTextualMode tactic uniformly switches the server content mode from multimedia to textual. A corresponding tactic switchToMultimediaMode achieves the opposite effect. An adjustServerPoolSize tactic increments or decrements the server pool size by an

integral count. Associated with each of these three tactics is a cost-benefit attribute vector, each consisting of the four previously described attributes, shown in Table 4.8.

Table 4.8: Znn.com tactic cost-benefit attribute vectors

Tactic	u_R	u_F	u_C	u_D
switchToTextualMode	-2 steps	-1 step	+0 (no change)	3
switchToMultimediaMode	+1 step	+1 step	+0	3
adjustServerPoolSize(int Δk)	-2 steps *if* $\Delta k > 4$, -1 step *if* $0 < \Delta k \leq 4$, +1 step *if* $\Delta k < 0$	+0	-1 step *if* $c(k + \Delta k) < Th_{bud}$, *else* +1 step	1

We simplify the illustration of strategy selection by defining two placeholder strategies, DropFidelityStrategy that uses the tactic switchToTextualMode, and EnlargeServerPoolStrategy that invokes the tactic adjustServerPoolSize with a Δk-argument of 5 (which yields a -2-step effect on the u_R dimension). This eliminates the step of calculating the aggregate attribute vectors (cf. Section 4.4.2) and focuses our discussion on the attribute- and utility-based strategy selection.

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, both strategies are applicable: one to change the content mode to textual and the other to increase the size of the server pool. So we need to score the strategies to determine which one is most appropriate given the utility preferences. The specified tactic attribute vectors would yield aggregate attribute vectors for the two strategies as shown in Table 4.9, followed by their weighted utility evaluation in Table 4.10.

Table 4.9: Znn.com aggregate attribute vectors for two applicable strategies

Strategy	u_R	u_F	u_C	u_D
DropFidelityStrategy	-2⇒low	-1⇒textual	+0⇒within	3
EnlargeServerPoolStrategy	-2⇒low	+0⇒multimedia	+1⇒over	1

Table 4.10: Znn.com utility evaluation for two applicable strategies

Strategy	Weighted Utility Evaluation
DropFidelityStrategy	$U = 0.4\,(1) + 0.2\,(0) + 0.3\,(1) + 0.1\,(0.5) = 0.75$
EnlargeServerPoolStrategy	$U = 0.4\,(1) + 0.2\,(1) + 0.3\,(0) + 0.1\,(1) = 0.70$

The utility scores indicate DropFidelityStrategy as the better adaptation strategy, given the current system conditions. Note that if Znn.com attributed a lower weight to budget, or 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. Although this example shows simple utility functions of a few points, one can define more complex utility functions and benefit from this utility-based technique.

4.6 Summary

In this chapter, we identified a set of language requirements to represent adaptation knowledge and presented the self-adaptation language, Stitch. We further motivated the required features by analyzing the task, knowledge, and cognitive model of a system administrator. We then discussed the concepts of quality dimensions and adaptation conditions, operator, tactic with cost-effect attributes, strategy, utility preferences, and strategy selection. Finally, we detailed the formal semantics of Stitch. The important features of the language and their traceability to the motivating requirements are summarized in Table 4.11. Next, we present the design of Rainbow framework design and its customization points..

Table 4.11: Traceability summary of Stitch language features

#	**Sys-Admin Task**	**Stitch Feature**	**Semantics Treatment**
1	Business qualities of concern	Quality dimensions	Schema of utility profile/functions
2	Improvement opp. indicator	Adaptation conditions	Acme constraints
3	Basic system-provided command	Operator	Architectural style operators
4	Step of adaptation action	Tactic	MDP actions + oper. semantics
4a	- conditions for applying action	- condition block	Acme predicates
4b	- sequence of commands	- action block	Operational semantics
4c	- outcome expected of action	- effect block	Acme predicates
5	Patterns of adaptation with condition-action-delay	Strategy	MDP subgraph + oper. semantics
5a	- choice of condition-action pair	- condition matching	MDP states + atomic propositions
5b	- uncertainty in action outcome	- branch probabilities	MDP transition probabilities
5c	- chosen action	- tactic action	MDP actions
5d	- observation of outcome	- delay time-window	MDP nondet. transition target s'
6	Factors in choosing adaptation	Cost-benefit attributes	Schema of attribute-utility relation
7	Biz. preferences over qualities	Utility preferences	Assignment of weights to utilities
8	Choice from strategy alternatives	Strategy selection	Algorithm for strategy selection

Chapter 5

Customizable Framework

Chapter 3 presented the overall Rainbow approach and described the high-level functionalities of the framework components, offering only a brief glimpse into the customization points. Chapter 4 presented the Stitch self-adaptation language in the context of the Rainbow framework. Having introduced the Rainbow customization points, we now provide more details to demonstrate how they reduce effort in engineering a system for self-adaptation.

In this chapter, we describe the design and engineering of the Rainbow framework, focusing particularly on the customization points and how the framework pieces fit together to facilitate Rainbow customization. We briefly present the architecture of the framework, then provide more details on each component to show how it functions and is customized. We showcase the approach with a full customization using the Znn.com example. Finally, we introduce the Rainbow Adaptation Integrated Development Environment (RAIDE).

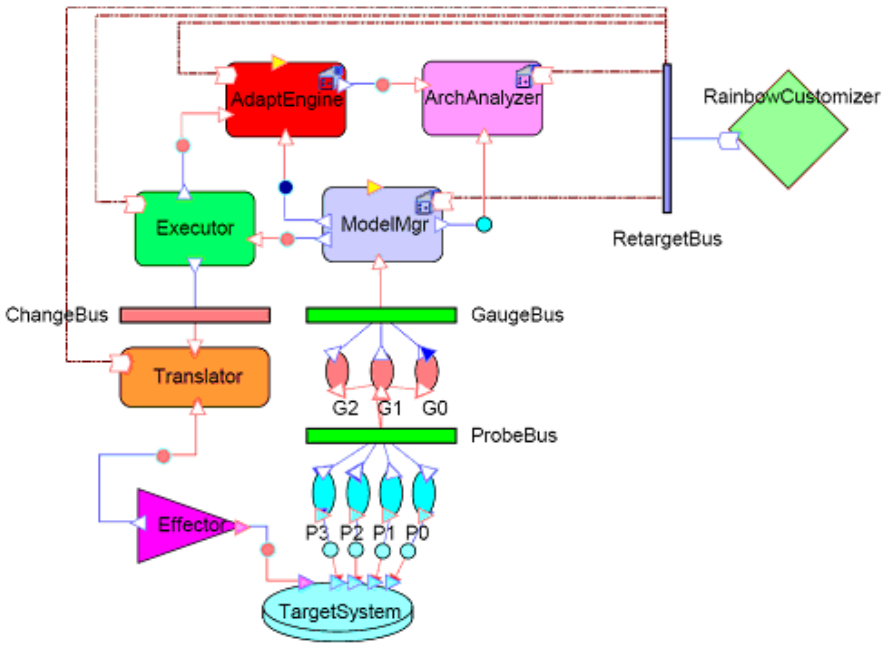


Figure 5.1: Rainbow architectural diagram

5.1 Architecture and Design of Rainbow

As we have argued, the Rainbow approach is to provide a generic, reusable framework that can be tailored to a specific architectural style of a target system and for adaptation with respect to specific quality dimensions. The C&C architecture of the Rainbow framework, described in Acme and modeled in a graphical architecture design environment called AcmeStudio [SG04], is diagrammed in Figure 5.1. Table 5.1 shows the descriptions of architectural types for the main-Rainbow family. The complete set of architectural types, including the base-Rainbow and the representation-implementation families, appear in Appendix A on page 173.

Table 5.1: Rainbow architectural style description — Main Family

Type Name	**Functional Description**
AdaptationEngineT	Performs adaptation
ArchAnalyzerT	Evaluates architectural constraints on the model
CustomizerT	Customizes Rainbow components
EffectorT	Propagates and enacts changes in the target system
ExecutorT	Carries out adaptation actions on system via translator
GaugeT	Updates model properties
ModelManagerT	Manages model instance(s), provides model info query
ProbeT	Extracts monitoring information from target system
TargetT	Represents or simulates the target system
TranslatorT	Maintains correspondence between model and system
EffectingConnT (*ControlCn*)	For Effector to effect changes on target system
ChangeNotifyConnT (*NotifyCn*)	Change notification bus, listen for system change requests
GaugeNotifyConnT (*NotifyCn*)	A gauge bus
ProbeNotifyConnT (*NotifyCn*)	A probe bus
RetargetConnT (*NotifyCn*)	Customization notification bus for Customizer
TriggeredAnalyzeConnT (*PollCn*)	Specific polling type, model update triggers analysis
ProbingConnT (*PollCn*)	Specific polling type, instruments the target system
EffectingProvPortT (*ProvideP*)	Enables effecting changes on target system
MonitoringProvPortT (*ProvideP*)	Enables target system states to be monitored
RetargetProvPortT (*ProvideP*)	Customization point to tailor component to specific style
EffectingReqPortT (*RequireP*)	On the Effector, effects system changes
MonitoringReqPortT (*RequireP*)	On the Probe, monitors (extracts) system states
RetargetReqPortT (*RequireP*)	For Customizer to tailor component to specific style
UserPrefReqPortT (*RequireP*)	For Rainbow to receive user preference info
RetargeteeRoleT (*SubscriberR*)	Event bus subscriber role played by retargetee component

There is a close correspondence between this architectural model and the Rainbow framework diagram in Figure 3.2 on page 29. This architecture provides a natural decomposition that (a) logically separates the customization points on individual components and (b) allows runtime separation of concerns where each component performs a cohesive self-adaptation function. The

notional framework customization points, the plug-in pieces, in the diagram are realized as Retarget Ports (RetargetProvPortT) on the major Rainbow components, joined by a Retarget Bus to the Rainbow Customizer component. In implementation, the Rainbow Customizer is realized by the Oracle Java class, which instantiates the Rainbow component objects and provides the initialization parameters to configure the components.

In this section, we describe the design of each Rainbow component and illustrate how each is customized, and by whom, using the Znn.com example. The complete customizations appear in Appendix C. A summary of who does what in what form is show in Table 5.2.

Table 5.2: Who-Does-What-How summary of framework customization

Artifact	Parts/Encapsulation	Who?	How?
Style	Types, rules, properties,	*Style*	Acme
.	operators	*Writer*	Signature
Effector	(System hook to make changes)	*System Adapter*	Desc. schema + impl.
Probe	(System hook to check properties)	*System Adapter*	Desc. schema + impl.
Gauge	Type, inst., model, properties	*Gauge Writer*	Desc. schema + impl.
Mapping	M_{oe}: operator→effector	*Adaptation*	Translation mapping
.	M_{gp}: gauge→probe	*Integrator*	Part of gauge spec
Tactic	Context: style (incl. the operators); script of operators, conditions, effects	*Tactic Writer*	Stitch Tactic syntax
Strategy	Context: style (types) + tactics set; applicability filter (style/domain); decisions: condition-action-delay	*Strategy Writer*	Stitch Strategy syntax
Utility	Cost-benefit attributes of tactics; utility functions; preference weights	*Adaptation Integrator*	Utility spec schema

5.1.1 Rainbow Deployment Architecture

Having described the Component-and-Connector view of the Rainbow architecture, we now discuss its deployment view, which addresses how the components are allocated to computing nodes of the running system. Each of the top-level Rainbow components is an *active* object. The framework defines an IRainbowRunnable interface, realized by a common super class AbstractRainbowRunnable, which manages the thread lifecycle, state transition, and disposal of the subclassed component. Each active Rainbow component extends AbstractRainbowRunnable.

Since the Architecture-Layer components of Rainbow share access to the architecture model via the Model Manager, running them within the same process eliminates the overhead of interprocess communication. The Rainbow Oracle is the "main" of the Rainbow runtime, i.e., the class where the Java main() method resides. Once executed on a Java VM, the Oracle sets up the event middleware, initializes all of the Architecture-Layer components, and creates a GUI console (a.k.a. *Rainbow Control Console*) for administrator oversight. Figure 5.2 illustrates the Rainbow Oracle GUI console and Rainbow's deployment architecture.

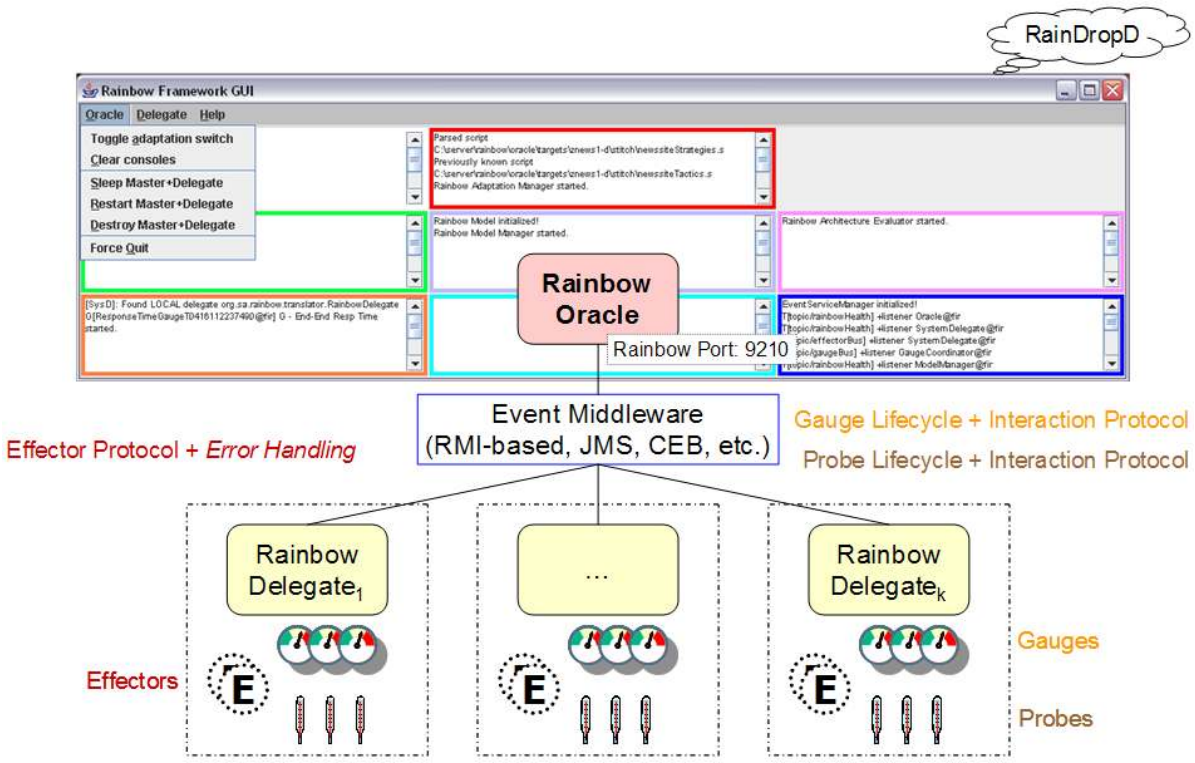


Figure 5.2: Rainbow run time deployment

The Translation Infrastructure, discussed in Section 5.1.3 below, is where the distribution of various Rainbow parts occurs. As the Rainbow architecture indicates, the Model Manager and the Strategy Executor each has a port that communicates via an event bus with the distributed parts of Rainbow. The idea is to deploy a *Rainbow Delegate*, a surrogate actor of the Rainbow runtime, on every computing node of the target system. Each Delegate sets up its end of the communication infrastructure and initializes the applicable probes, gauges, and effectors on its node. It serves as a locus of communication between the Oracle and the gauges and effectors on the Delegate's node.

Resource overhead incurred by the Delegate is minimal. Data from actual runs indicates each Delegate consuming less than 2% of CPU (but occasionally sustained at ~5-10%) with a couple megabytes of memory footprint, and this overhead is further reducible with optimization in implementation. Nevertheless, on a computing node with highly constrained memory and CPU resources, the adaptation engineer may choose to deploy only probes on the node and wire the probes to report on the Probe Bus via the Oracle or a neighboring Delegate node.

Finally, to facilitate failure recovery (i.e., restart), remote management, binary deployment, and version update, each of the Oracle and Delegate processes is spawned by a RainDropD daemon. This daemon is required on each node to deploy Rainbow. Subsequently, the administrator could manage all Delegate instances from a central console. The RainDropD daemon automatically respawns its Delegate if the Delegate crashes. The daemon allows the administrator to issue *stop*, *restart*, and *terminate* commands to each Delegate. It also allows rolling out version updates of the Rainbow software and library updates of probes, gauges, and effectors. As a basic

security measure, the daemon responds only to messages from the Oracle node.

In a Rainbow-properties file, the *adaptation integrator* defines the configuration parameters to initialize the Oracle and Delegates. The properties are loaded with one-time variable substitutions by the singleton org.sa.rainbow.Rainbow class, which provides access to the common Rainbow properties. The excerpt below illustrates configuration for host locations, logging, event infrastructure, and various Rainbow component-specific settings. The *master* host refers to the host on which Oracle runs. The *Rainbow service port* is the common TCP/IP port used for event communication and remote connection with the Rain Drop Daemon. To prevent name clash, each Delegate defines a unique *Delegate identifier* of "name@location" when communicating with the Oracle, and establishes a *beacon period* for liveness checking. Component-specific settings define the file locations for the customization files described in this section. Finally, system configuration tells Oracle the locations of all target system nodes. A more sophisticated implementation, e.g., one based on an advanced event infrastructure, would allow the nodes in the system to be discovered on Rainbow initialization.

```
1  ### Utility mechanism configuration
2  #- Config for Log4J, with levels: OFF,FATAL,ERROR,WARN,INFO,DEBUG,TRACE,ALL
3  logging.level = INFO
4  ...
5  ### Rainbow component customization
6  ## Rainbow host info and communication infrastructure
7  #- Location information of the master and this deployment
8  rainbow.master.location.host = localhost
9  #- Location information of the deployed delegate
10 rainbow.deployment.location = ${rainbow.master.location.host}
11 #- Rainbow service port
12 rainbow.service.port = 9210
13 ...
14 #- Event infrastructure, type of event middleware: rmi | jms | que
15 rainbow.event.service = rmi
16 ...
17 ## RainbowDelegate and ProbeBusRelay configurations
18 rainbow.delegate.id = RainbowDelegate@${rainbow.deployment.location}
19 rainbow.delegate.beaconperiod = 5000
20 rainbow.delegate.startProbesOnInit = false
21 probebus.relay.id = ProbeBusRelay@${rainbow.deployment.location}
22 ...
23 ## Model Manager customization
24 customize.model.path = model/ZNewsSys.acme
25 ...
26 ## Translator customization
27 customize.gauges.path = model/gauges.yml
28 customize.probes.path = system/probes.yml
29 customize.archop.map.path = model/op.map
30 customize.effectors.path = system/effectors.yml
31 ## Adaptation Manager
32 customize.scripts.path = stitch
33 customize.utility.path = stitch/utilities.yml
34 customize.utility.trackStrategy = uSF
35 customize.utility.scenario = scenario 2
36
37 ## System configuration information
38 customize.system.target.0 = ${rainbow.master.location.host}
39 customize.system.target.1 = oracle
40 customize.system.target.size = 2
```

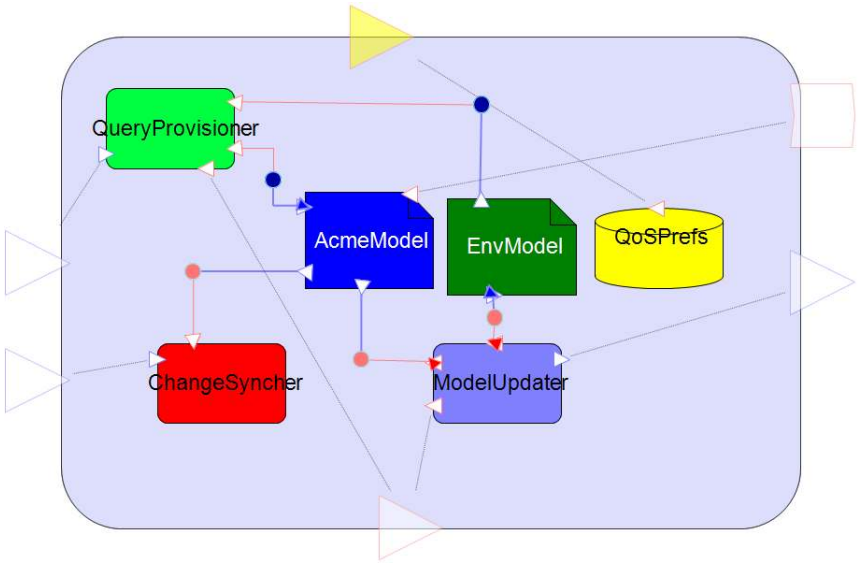


Figure 5.3: Architectural decomposition of the Model Manager

5.1.2 Model Manager and Rainbow Models

As introduced in Section 3.3.3, the *Model Manager* maintains and manages instances of the architecture model of the target system. It also maintains information about the system's execution environment—the environment model, including spare computation resources, network resources, contextual entities—and references between environment elements and architectural elements. The architectural decomposition of the Model Manager is shown in Figure 5.3.

The Model Updater component acts as the Gauge Consumer (Figure 3.5), consuming gauge updates via the Gauge Bus (discussed below in Section 5.1.3). Updates to the Acme Model occur whenever a gauge event is consumed. The Query Provisioner serves two purposes: It extracts the relevant and available query information from the Acme Model and the Env Model. If a property is defined in the model, but the current value is not known, then it is the duty of the Query Provisioner to issue a system query through the query mechanism of the gauge infrastructure. The Change Syncher represents a feature to keep track of changes announced by the Executor on a separate Acme Model instance. In future work, this allows tracking intended changes against observed system changes and provides a placeholder for adaptation learning.

The user-oriented port to update the QoS Preferences is achieved by adding or modifying design rules in the Acme Model, but future work could allow dynamic, user task-based input (see Section 9.2.2). The Acme Model and Env Model data components are embodied in the implementation Java class, RainbowModel. These two Model components extend existing implementation on AcmeLib for manipulating architecture models defined in Acme [CMU08]. Finally, the three Provide (component side edges) and one Require (bottom edge) ports of the Model Manager share a common Java interface, Model, with specific methods for model update (updateProperty (String iden, String value)), constraint checking (evaluateConstraints ()), and query (getProperty (String id)).

The customization point of the Model Manager is a pair of Rainbow Models: The architecture

model consists of Acme descriptions of one or more styles, defined by the *style writer*, and an instance that extends them. The environment model likewise consists of Acme descriptions of the environment style and instance. The example below shows the ZNewsFam style with a few types shown and other details elided. Some properties of ServerT are updated by gauges, and the design rules in ClientT are checked by the Architecture Evaluator.

```
1  import TargetEnvType.acme;
2
3  Family ZNewsFam extends EnvType with {
4      Component Type ServerT extends ArchElementT with {
5          Property deploymentLocation : string << default : string = "localhost"; >>;
6          Property load : float << default : float = 0.0; >> ;
7          Property reqServiceRate : float << default : float = 0.0; >> ;
8          Property byteServiceRate : float << default : float = 0.0; >> ;
9          Property fidelity : int << HIGH:int = 5; LOW:int = 1; default:int = 5; >>;
10         Property cost : float << default : float = 1.0; >> ;
11         Property lastPageHit : Record [uri : string; cnt : int; kbytes : float; ];
12         rule anotherConstraint = heuristic self.load <= MAX_UTIL;
13     }
14     Component Type ClientT extends ArchElementT with {
15         Property deploymentLocation : string << default : string = "localhost"; >>;
16         Property experRespTime : float << default : float = 100.0; >> ;
17         Property reqRate : float << default : float = 0.0; >> ;
18         rule primaryConstraint = invariant self.experRespTime <= MAX_RESPTIME;
19         rule reverseConstraint = heuristic self.experRespTime >= MIN_RESPTIME;
20     }
21     ...
22     Properties { ... }  // see ZNewsSys definition below
23 }
24
25 System ZNewsSys : ZNewsFam = {
26     Property MIN_RESPTIME : float = 100.0;
27     Property MAX_RESPTIME : float = 1000.0;
28     ...
29     Property TOLERABLE_PERCENT_UNHAPPY : float = 0.4;
30     Property MIN_UTIL : float = 0.1;
31     Property MAX_UTIL : float = 0.75;
32     Property MAX_FIDELITY_LEVEL : int = 5;
33     Property THRESHOLD_FIDELITY : int = 2;
34     Property THRESHOLD_COST : float = 4.0;
35     Component s0 : ServerT = new ServerT extended with {
36         Property deploymentLocation = "oracle";
37         Property cost = 0.9;
38         Property fidelity = 3;
39         Property load = 0.198;
40         ...
41     }
42     Component c0 : ClientT = new ClientT extended with {
43         Property deploymentLocation = "127.0.0.1";
44         Property experRespTime = 2360.2585;
45         ...
46     }
47     Connector conn : HttpConnT = new HttpConnT extended with {
48         Property latencyRate = 0.0;
49         Role req  = { Property isArchEnabled = true; }
50         Role rec  = { Property isArchEnabled = true; }
51     }
52     ...
53     Attachments { ... }
54 }
```

5.1.3 Translation Infrastructure—Monitoring and Action

To support a broad spectrum of systems and achieve **generality** (Section 3.1), the Translation Infrastructure serves the dual role of decoupling and providing a bridge between the framework Architecture Layer and the platform and implementation of the System Layer. *Indirection* and *common communication protocols* are two effective design techniques to achieve decoupling, and *distribution* is an essential feature because most modern systems are distributed to varying degrees. For the Translation Infrastructure, correspondence mappings provide indirection, while an event middleware implementing common communication protocols facilitates the distribution of probes, gauges, and effectors on heterogeneous computing nodes of the target system.

As highlighted in Section 3.3.2, the *Translation Infrastructure* consists of *probes*, *gauges*, *effectors*, and correspondence mappings, which together comprise its customization point. In the Rainbow architecture, this infrastructure encompasses four component types, six connector types, and related interfaces. All translation components share the correspondence mappings of elements, operators, properties, and errors maintained by the Translator component. Generation of the maps and boot-strapping of the initial architecture model are not addressed in this thesis, but may be provided by an architecture discovery system such as DiscoTect [YGS+04].

For monitoring, ProbeT component instances instrument the target system via ProbingConnT polling connector instances and then publish the information on the ProbeNotifyConnT connector instance (Probe Bus). For instance, a Link Latency Probe periodically polls the link latency between two IP nodes, src_ip and tgt_ip, in the target system and publish that link latency on the Probe Bus. Multiple Link Latency Probes might be deployed to monitor different IP pairs.

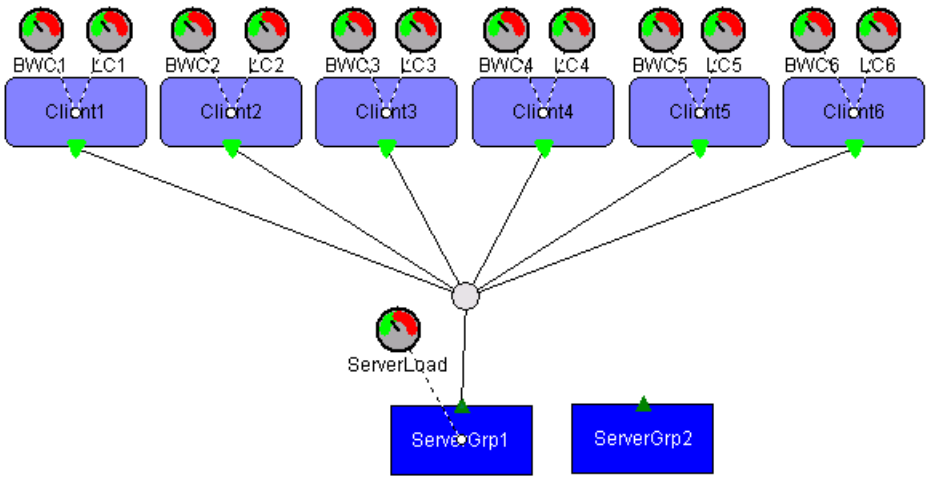


Figure 5.4: Visual connection of gauges to the architecture model

GaugeT component instances subscribe to relevant probes on the Probe Bus to aggregate and abstract system information, then publish the results on the GaugeNotifyConnT connector instance (Gauge Bus). At the subscriber end of the Gauge Bus is the Model Manager, which has gauge configuration knowledge and thus serves both as the Gauge Manager that creates gauges and manages gauge lifecycles, and as the Gauge Consumer that consumes gauge reports and updates corresponding properties in the architecture model. For example, a Latency Gauge

collects the latency output from a Link Latency Probe, averages the values over a configurable time window, and publishes the average latencies for the Model Manager to update the latency property of a connector with corresponding source and target IP addresses src_ip and tgt_ip. Note that depending on gauge configuration, one Latency Gauge might handle one or multiple Link Latency Probes. Figure 5.4 illustrates how gauges attach to the model.

For action, a change operation propagates via the ChangeNotifyConnT connector instance (Change Bus) and is translated by the Translator into system level operation(s) and issued to the EffectorT component instance. The Effector connects via an EffectorConnT connector instance to effect changes on the target system.

The deployment and distribution of gauges, probes, and effectors on target system nodes can vary in scheme for different reasons, including platform heterogeneity and differences of resource constraints between target computing nodes. We therefore provide a generic event mechanism that is flexible for different deployment needs. Our generic event transport mechanism realizes a common interface and allows specific event implementations to be plugged in (e.g., RMI-based, Java Messaging System), which can be configured by setting the rainbow.event.service property of the Rainbow customization file with an event-service identifier (e.g., "rmi", "jms", "que"). This design enables Rainbow components to interact with different target systems that communicate using different event mechanisms. We have also designed a standard set of communication protocols to support pluggability of probes and gauges into Rainbow. In the following paragraphs, we present the design of the probe and gauge communication protocols and describe an integrating Translator. Together, they enable distribution, indirection, and decoupling.

Probe Protocol Design The probe protocol enables interaction between the Translator as the Probe Manager, gauges that use probes, and the probes themselves. Each Rainbow Delegate serves as the local proxy for the Probe Manager in managing probes local to the Delegate host; it also propagates events between local probes and the Translator and gauges. Four core event types are defined in the probe event vocabulary: ID:string ::= "name@location", Capabilities:seq<string> ::= <...>, Target:string ::= "location", and Period:long ::= [0-9]+ /*ms*/. Note that ID already encodes the probe target. Four probe states are defined: *CREATED*, *ACTIVE*, *INACTIVE*, *DESTROYED*. The protocol rules are as follows:

1. Probe initialization: a probe can be spawned manually or by Rainbow
   (a) Probe connects to the probe bus (possibly via a probe bus relay)
   (b) Probe announces attributes: ID, capabilities (i.e., type), probe target, update period
   (c) Probe enters *CREATED* state, initialized and ready to announce properties
2. While *not DESTROYED*, a probe periodically announces a "beacon event" of ID and type
3. Based on the probe type and probe target, the Probe Manager makes a probe match[1] and issues command via the probe bus to activate it; probe enters *ACTIVE* state

[1]Conceptually, the architecture model contains the knowledge of what property is associated with which probes in the system. Practically, this information most likely lives in the translation infrastructure, known in a distributed manner to the gauges. In effect, a gauge would know with which probe(s) it should interact.

4. While *ACTIVE*, a probe periodically announces on the probe bus the target properties under its observation; announcements take the form: "[timestamp] {name/alias} msg"; or it could be deactivated via the probe bus to stop publishing and enter *INACTIVE* state
5. A probe could be activated and deactivated any number of times
6. While either *ACTIVE* or *INACTIVE*, a command could issue from the probe bus to publish the currently known property values, and the matching probe would immediately publish its latest known value; this is used by the gauge to query values on-demand
7. While *INACTIVE*, a command could issue from the probe bus to destroy the probe, which causes the probe to be terminated and/or garbage-collected; the probe enters *DESTROYED* state (though state is not persisted); once *DESTROYED,* a probe may not be re-activated

Gauge Protocol Design For the gauge protocol, we build on work done elsewhere on a standard Gauge Infrastructure [GSC01]. The gauge protocol enables interaction between the Gauge Manger at the Oracle end, the Rainbow Delegates as local Gauge Manager proxies, and the gauges themselves. Here, we describe a simplified version, starting with a single event type: ID:string ::= "name@location". Like probes, four gauge states are defined: *CREATED*, *ACTIVE*, *INACTIVE*, *DESTROYED*. The protocol rules are defined as follows:

1. Rainbow Delegates connects to the gauge bus from remote machines, ready to serve requests from Gauge Manager
2. Gauge Manager reads gauge specification and coordinates gauge creation by delegating the request to designated Rainbow Delegates
3. Rainbow Delegates create gauges locally and return identifiers for the created gauges
4. Gauge Manager keeps track of all the created gauges, and serves any request for gauge reference by returning the gauge UID
5. Gauge Manager configures each gauge as needed and activates gauge
6. Gauge Manager deletes a gauge if it's no longer needed

IGauge defines the public interface of, and provides a handle to, a gauge. A static class, GaugeProtocolHandler, is provided to process gauge events, using gauge protocol identifiers and strings defined in the IGaugeProtocol interface. Every event message is defined as a set of key-value pairs, with meta-attributes that may be specific to event mechanisms. For example, JMS defines the meta-attributes: ID to uniquely identify the message, ReplyTo for response message, Type of the message, MessageID, Destination, Expiration, Priority, and Timestamp.

Translator Design In addition to "listening" to the change bus to translate operators into effector operations, the Translator also "listens" to the probe bus and serves as a Probe Manager and Registry. Probes can register and deregister themselves with the Translator. Gauges can lookup, subscribe to, and unsubscribe from probes through the Translator.

Customization of the Translation Infrastructure entails specifying the probes, gauges, and effectors. The probe specification, composed by the *system adapter*, defines the available probes

for the target system. The excerpt below illustrates two example probe instances with typical parameters. (Key-value string substitutions are allowed, as illustrated on lines 2 and 19 below.) Specifically, PingRTTProbe1 is a probe of type (*alias*) "pingrtt," to be deployed on a node (*location*) defined by the variable "customize.system.target.1," implemented in Java (*type*) by the *class* org.sa.rainbow.translator .znews.probes.PingRTTProbe, with a report *period* of 1500 ms and Java arguments supplied via *args*. LoadProbe1 shares similar probe parameters, but is implemented by a shell script located as defined in *path*, with argument supplied via *argument*.

```
1  vars:
2    _probes.commonPath: "${rainbow.path}/system/probes"
3  probes:
4    PingRTTProbe1:
5      alias: pingrtt
6      location: "${customize.system.target.1}"
7      type: java
8      javaInfo:
9        class: org.sa.rainbow.translator.znews.probes.PingRTTProbe
10       period: 1500
11       args.length: 1
12       args.0: "${rainbow.master.location.host}"
13       args.1: "${customize.system.target.2}"
14   LoadProbe1:
15     alias: load
16     location: "${customize.system.target.1}"
17     type: script
18     scriptInfo:
19       path     : "${_probes.commonPath}/loadProbe.pl"
20       argument: "-k -s"
```

The gauge specification, composed by the *gauge writer*, defines available gauges. The excerpt below illustrates one gauge type, ResponseTimeGaugeT whose parameters are as those described in Section 3.3.2. Specifically, EERTG1 is a gauge instance of *type* ResponseTimeGaugeT that reports an end-to-end response time *value* "end2endRespTime" to the host "delegate.oracle," meant to update the property "c0.experRespTime" of an Acme model named ZnewsSys, to be deployed on a node (*targetIP*) defined by the variable "customize.system.target.0," with a heartbeat (*beaconPeriod*) of period 30000 ms, implemented by the Java class org.sa.rainbow.translator .znews.gauges.End2EndRespTimeGauge, and configured to sample values from the target probe type "clientproxy" with a period of 1000 ms.

```
1  gauge-types:
2    ResponseTimeGaugeT:
3      values:
4        end2endRespTime : double
5      setupParams:
6        targetIP:
7          type:    String
8          default: "localhost"
9        beaconPeriod:
10         type:    long
11         default: 30000
12       javaClass:
13         type:    String
14         default: "org.sa.rainbow.translator.znews.gauges.End2EndRespTimeGauge"
15     configParams:
16       samplingFrequency:
17         type:    long
18         default: 1000
19       targetProbeType:
20         type:    String
```

```
21          default: ~
22      comment: "Reports end-to-end response time from client proxy."
23  gauge-instances:
24    EERTG1:
25      type:   ResponseTimeGaugeT
26      model: "ZNewsSys:Acme"
27      mappings:
28        "end2endRespTime(delegate.oracle)" : c0.experRespTime
29      setupValues:
30        targetIP: "${customize.system.target.0}"
31      configValues:
32        samplingFrequency: 1000
33        targetProbeType   : clientproxy
34      comment: "EERTG1 is associated with client component of ZNewsSys in Acme"
```

The effector specification, composed by the *system adapter*, defines available effectors in the target system. The excerpt below illustrates one effector with typical parameters. Specifically, setFidelity1 is an effector available on a node (*location*) defined by the variable "customize.system.target.1," and implemented by shell script located as defined in *path*, with argument supplied via *argument*.

```
1  vars:
2    _effectors.commonPath: "${rainbow.path}/system/effectors"
3  effectors:
4  # test from GUI with <host>, SetFidelity, fidelity=<1|3|5>
5    setFidelity1:
6      location: "${customize.system.target.1}"
7      type: script
8      scriptInfo:
9        path     : "${_effectors.commonPath}/changeFidelity.pl"
10       argument: "-l {fidelity}"
```

5.1.4 Architecture Evaluator

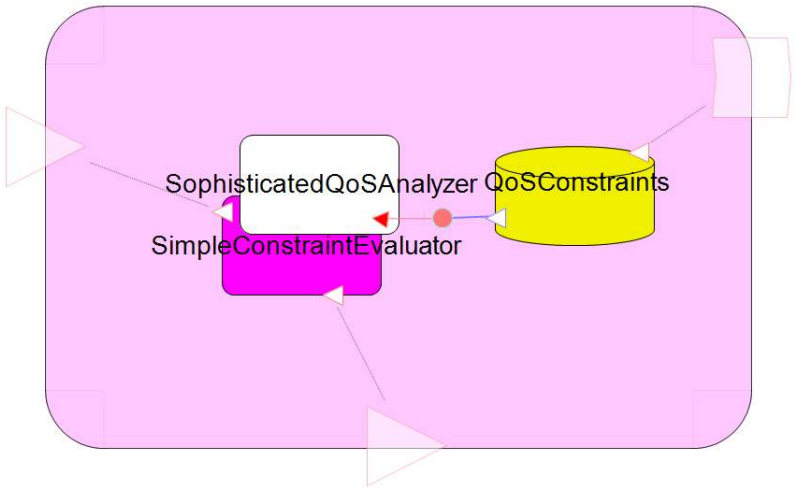


Figure 5.5: Architectural decomposition of the Architecture Evaluator

The *Architecture Evaluator*, as introduced in Section 3.3.4, evaluates the conformance of the architecture model to a predefined set of design rules, as either invariants or heuristics. The

style writer customizes the Architecture Evaluator by specifying design rules in the architecture model, as already illustrated by the Acme description in Section 5.1.2.

The architectural decomposition of the Architecture Evaluator is shown in Figure 5.5. The QoS Constraints data component represents the rules described in the Acme architecture model. In the current design, the Architecture Evaluator acts as a Simple Constraint Evaluator and invokes evaluateConstraints() on the architecture model to determine if any constraint violation has occurred. We discuss its design alternatives in Section 9.2.2.

5.1.5 Adaptation Manager

The *Adaptation Manager*, as introduced in Section 3.3.5, responds to a constraint violation and selects the best strategy to fix the current problem conditions. For the adaptation process, it maintains (a) a repertoire of Stitch strategies and tactics with the associated cost-benefit attribute vectors, and (b) the utility profiles and preferences. These comprise its customization point.

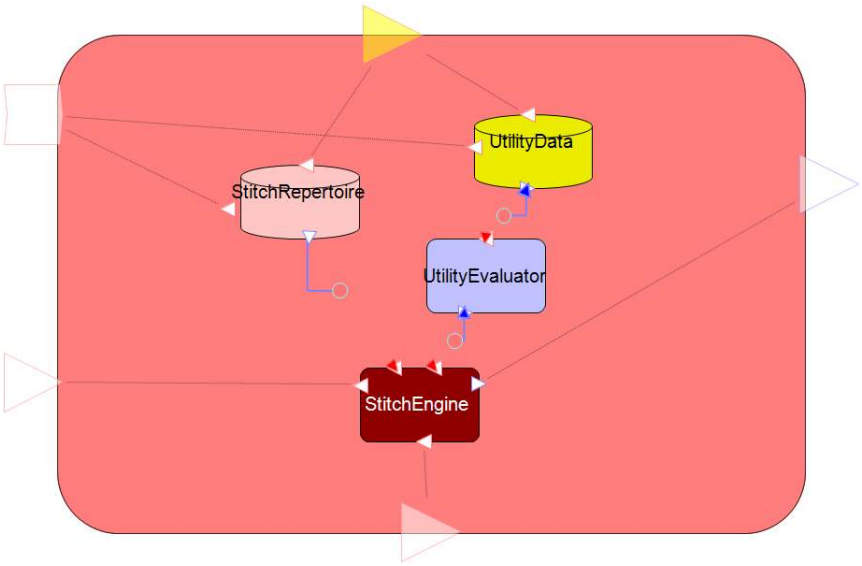


Figure 5.6: Architectural decomposition of the Adaptation Manager

The architectural decomposition of the Adaptation Manager is shown in Figure 5.6. The Stitch Repertoire data component holds the repertoire of strategies and tactics; the Utility Data component stores the utility profiles and preferences. Users update the Stitch Repertoire and Utility Data via the User Preference port.

A Stitch Engine awaits adaptation trigger from the Architecture Evaluator, which sets a flag via the blue Provide port to perform adaptation in the next run cycle. Upon trigger, the Stitch Engine queries the Rainbow Model for the context of the constraint violation via getAcmeModel() (the bottom-edge Require port on the Stitch Engine component). It then searches the Stitch Repertoire for applicable strategies, requests the Utility Evaluator to select the best strategy, and enqueues the strategy with the Executor to effect it on the target system (left-edge Require port of Stitch Engine). In implementation, each of the Adaptation Manager subcomponents is realized by a Java class or method accessible to the AdaptationManager class.

The *tactic writer* defines tactics and *strategy writer* composes strategies in the Stitch syntax to produce adaptation scripts. The excerpt below illustrates one tactic and one strategy. A script defines a namespace with the keyword *module* and may import one or more namespaces. The tactic script imports (line 2) the namespace of the architecture model from the Acme description "ZNewsSys.acme," renaming references to the family as *T* and instance as *M*, (line 3) the set of operators, and (line 4) Stitch utility libraries. The strategy script imports the tactic script and acquires its namespace in the process.

```
1  module newssite.tactics;
2  import model "ZNewsSys.acme" { ZNewsSys as M, ZNewsFam as T };
3  import op "znews1.operator.ArchOp" { ArchOp as S };
4  import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
5
6  // Enlist n free servers into service pool.
7  tactic enlistServers (int n) {
8      condition {
9          // some client should be experiencing high response time
10         exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
11         // there should be enough available server resources
12         Model.availableServices(T.ServerT) >= n;
13     }
14     action {
15         set servers = Set.randomSubset(Model.findServices(T.ServerT), n);
16         for (T.ServerT freeSvr : servers) {
17             S.activateServer(freeSvr);
18         }
19     }
20     effect {
21         // response time decreasing below threshold should result
22         forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
23     }
24 }
```

```
1  module newssite.strategies;
2  import lib "newssiteTactics.s";
3
4  define boolean styleApplies =
5      Model.hasType(M, "ClientT") && Model.hasType(M, "ServerT");
6  define boolean cViolation =
7      exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
8
9  // While it encounters high experienced response time, this simple strategy
10 // first enlists one new server, then lowers fidelity one step, then quits
11 strategy SimpleReduceResponseTime
12 [ styleApplies && cViolation ] {
13   t0: (cViolation) -> enlistServers(1) @[1000 /*ms*/] {
14     t1: (!cViolation) -> done;
15     t2: (cViolation) -> lowerFidelity(2, 100) @[3000 /*ms*/] {
16       t2a: (!cViolation) -> done;
17       t2b: (default) -> TNULL;  // in this case, we have no more steps to take
18     }
19   }
20 }
```

Composed by the *adaptation integrator* from business input, the utility specification defines the utility profiles of the quality dimensions and the business preferences over them. The excerpt below illustrates a set of utility profiles, a set of preference weights, and a set of tactic cost-benefit attribute vectors. The dimension *uR* defines the utility profile for Average Response Time, which is related to the property ClientT.experRespTime in the architecture, and has the

enumerated discrete pairs of domain and range values. When evaluating the utility function, range values are extrapolated for domain values that do not coincide with the defined points.

The cost-benefit attribute vector indicates that tactic "enlistServers" has a impact of -1000 ms on the *uR* dimension, none on the *uF* dimension, and +1.0 unit on the *uC* dimension. The weights distributed across the defined dimensions must sum to 1. Note that attribute vectors define delta impact to each utility dimension. When computing the utility value for a dimension in terms of delta impact, the aggregate average of the architectural property ClientT.experRespTime is queried from the model and added to the delta before evaluating the utility function.

```
1  utilities:
2    uR:
3      label: "Average Response Time"
4      mapping: "[EAvg]ClientT.experRespTime"
5      description: "R, client experienced response time (ms), float arch property"
6      utility:
7        0: 1.00
8        100: 1.00
9        200: 0.99
10       500: 0.90
11       1000: 0.75
12       1500: 0.50
13       2000: 0.25
14       4000: 0.00
15   ...
16 weights:
17   scenario 1:
18     uR: 0.3
19     uF: 0.4
20     uC: 0.2
21     uSF: 0.1
22   ...
23 vectors:
24 # Utility: [v] R; [^] C; [<>] F  (R by 1000 ms, increase C by 1 unit)
25   enlistServers:
26     uR: -1000
27     uF: 0
28     uC: +1.00
29   ...
```

5.1.6 Strategy Executor

The *Strategy Executor*, as introduced in Section 3.3.6, executes a strategy and its internal tactics on the target system. In particular, it handles the traversal of the strategy tree, the execution of each tactic, and the re-observation of the system via queries to the Rainbow Model. Customization is achieved by providing the architectural operators defined in the architectural style of the target system, along with a mapping of the operators to effectors. In implementation, the executor maps each architectural operator to a corresponding system-level effector via the Translator, which executes the effector and returns the result or any exception. The execution of an architectural operator in the tactic is passed to the Translator via synchronous, direct invocation; we discuss this design option in Section 8.2.

The operator mapping specification, also composed by the *adaptation integrator*, defines the mapping of style operators to effectors for the target system. The mapping description below illustrates three operators mapped to three corresponding effectors by name. A more sophisti-

cated implementation would also specify mapping of parameters and any return values, as well as potential errors.

```
1  setFidelity: changeState
2  activateServer: start
3  deactivateServer: stop
```

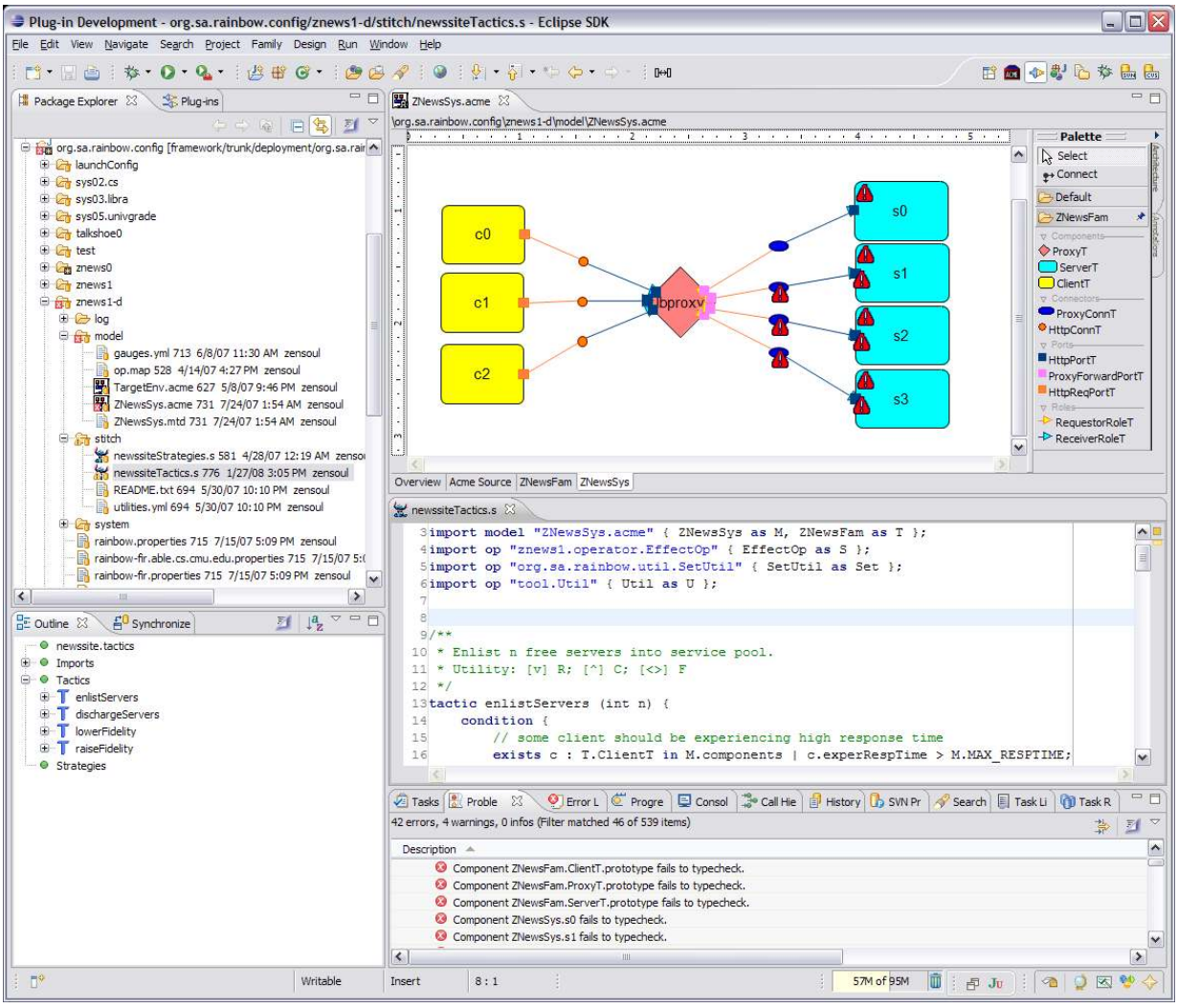


Figure 5.7: Mock-up of a Rainbow AIDE workbench

5.2 Adaptation Integrated Development Environment

The Rainbow framework described thus far focused mainly on the runtime components and the customization elements of Rainbow. In this section, we describe two additional components—the Stitch Script Editor and the Rainbow Control Console—that are important to the functionality, usability, and management of the Rainbow runtime. Together, the Rainbow framework runtime

and the Rainbow Control Console comprise a *Rainbow Development Toolkit*, which has been packaged as a distributable Eclipse project. The Stitch Script Editor is packaged by itself as an Eclipse plug-in. Together with AcmeStudio, an application also built in Eclipse, we are only steps away from an *Adaptation Integrated Development Environment* that integrates the Rainbow framework runtime and various editors. This AIDE would allow an adaptation engineer to edit Acme Models, write Stitch scripts, compose customization files, test adaptation runs, and deploy Rainbow runtime, all on one workbench, as illustrated by the mock-up in Figure 5.7.

5.2.1 Stitch Script Editor

With the help of Ali Almossawi, we provide an Eclipse-based script editor (shown in Figure 5.7) for the Stitch language with basic syntax highlighting, structure outline, and keyword completion to facilitate development of strategies and tactics. The Eclipse text editor API provides a marker mechanism for marking lines with information, warning, or error flags. The Stitch editor hooks into a back-end Stitch parser to support on-save script parsing, and reports syntax errors using the editor markers. Additional engineering effort would provide a more featured Stitch editor, with support for incremental build, navigation, linkage to architecture and model operators, code completion, and debugging execution.

5.2.2 Rainbow Control Console

The Rainbow Control Console, also known as the Oracle GUI (see Figure 5.2), provides a central console for managing the Rainbow runtime. In the current implementation with basic features, the console consists of eight output textareas and menu items for controlling local and remote Delegates, pushing out software updates, testing effectors, and deploying or undeploying probes.

Four of the eight consoles display the output of corresponding Rainbow components (note similar border colors as in the framework diagram). In addition, the lower-right textarea displays events on the Rainbow event infrastructure. The lower-center textarea displays probed states of the target system. The lower-left textarea displays Translator information, including the probes and effectors. The upper-left textarea displays output from the GUI console itself. Numerous enhancements are conceivable for the Rainbow Control Console, including greater control of each Rainbow component and more fine-grained control for probe, gauge, and effector deployment.

Summary

In this chapter, we described the architecture and design of the Rainbow framework, focusing specifically on the customization points of the Rainbow components. We discussed how to customize each component, who contributes the content, what customization looks like using examples from the Znn.com system. To support the Rainbow adaptation engineering process, we presented a Rainbow Adaptation Integrated Development Environment, which consists of the Rainbow framework runtime, the Stitch Script Editor, and the Rainbow Control Console. In the next two chapters, we present five example systems applying Rainbow, and evaluate how the Rainbow approach and framework achieve the thesis claims.

Chapter 6

Examples and Supporting Evidence

In Section 1.4.1, we described an evaluation plan for three thesis claims: **generality** to a broad spectrum of styles for common quality dimensions that modern architects are concerned with; **cost-effectiveness** to engineer and evolve self-adaptive systems relative to existing, specialized solutions; and **transparency** to make the adaptation process understandable, actions composable, and adaptation choice automatable for routine system adaptations. These claims lead to the validation points below:

V1 Demonstrate self-adaptation for a single property in three styles (**generality**)

V2 Demonstrate self-adaptation for at least three properties in one style (**generality**)

V3 Demonstrate use of environment model in at least one instantiation (**generality**)

V4 Demonstrate infrastructure reuse and ease of customization (**cost-effectiveness**)

V5 Demonstrate trade-off of multiple objectives in at least one instantiation (**transparency**)

In this chapter, we present five applications of Rainbow and two substantive system administrative examples from the computing infrastructure of Carnegie Mellon University. Together, these examples provide validation evidence for Rainbow. The first three sections describe the experimental systems from the exploratory phase of developing the Rainbow approach [CGS+02a, GCS03, CHG+04, GCH+04]. These demonstrate the feasibility of tailoring Rainbow to different styles (client-server and service-coalition) of system for more than one quality dimensions (*performance*, *cost*, and *security*). Sections 6.4 and 6.5 present two instantiations of the engineered Rainbow framework, one on the server infrastructure of TalkShoe, Inc., and the other on a news-service system, Znn.com. Finally, Sections 6.6 and 6.7 present qualitative evidence from cases of real system administration that support the Stitch self-adaptation language design. In Chapter 7, we recap how well the Rainbow approach fulfills the thesis claims.

6.1 Basic Client-Server System

The first Rainbow prototype aimed for a proof-of-concept framework to demonstrate the essential features of Rainbow on a basic client-server-style system, providing evidence that an adaptation cycle could adapt a system for a single quality dimension, *performance*, within a minute.

In this prototype, we instrumented source code to insert probes, used Remos for network monitoring [DGK+01], prototyped a gauge infrastructure, developed a few gauges, then hooked them to AcmeStudio serving simultaneously as the model manager, architectural evaluator, and model visualizer. We implemented an adaptation manager with built-in strategies and a simple table-driven translator to translate the architecture-level operators into system-level effectors to change the system. Details can be found in [CGS+02a, CGS+02b, GCS03, GCH+04].

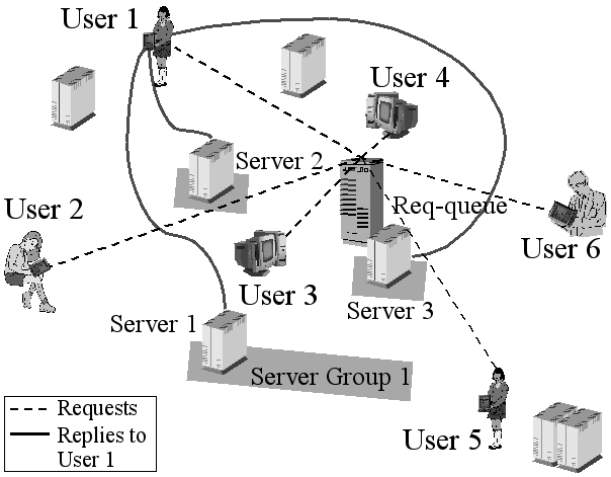


Figure 6.1: A class of web-based client-server systems

Consider the problem of providing load-balancing self-adaptation to improve the performance of a class of web-based client-server systems, illustrated in Figure 6.1, by minimizing the request-response latency that the users experience. The web-based system consists of a set of client processes making stateless, asynchronous file requests from a few clusters of servers via HTTP communication. Implemented in Java, each client that is connected to a server group sends requests to that group's shared request queue, and servers that belong to the group consume and fulfill requests from the queue. Component implementation provides remote method invocation (RMI) interface for effecting adaptation operations. The system concern focuses primarily on performance, specifically, the end-to-end request-response latency observed by the client process. We identify latency, server load, and available connection bandwidth as the important properties to monitor.

Based on the performance concern and system analysis, we define a client-server style for the system with *performance* properties. The architecture model, as illustrated in Figure 6.2, captures the client-server-style system with a fanning connector from one server-group component to a set of client components. Within each server-group component—in its Acme *representation*—is a number of replicated server components. Two spare server-group components are shown. The major elements of the style include:

- Types: ClientT, ServerT, ServerGroupT, LinkT
- Properties: ClientT.responseTime, ServerT.load, ServerGroupT.load, LinkT.bandwidth
- Architectural operators: ServerGroupT.addServer(), ClientT.move(ServerGroupT toGrp)

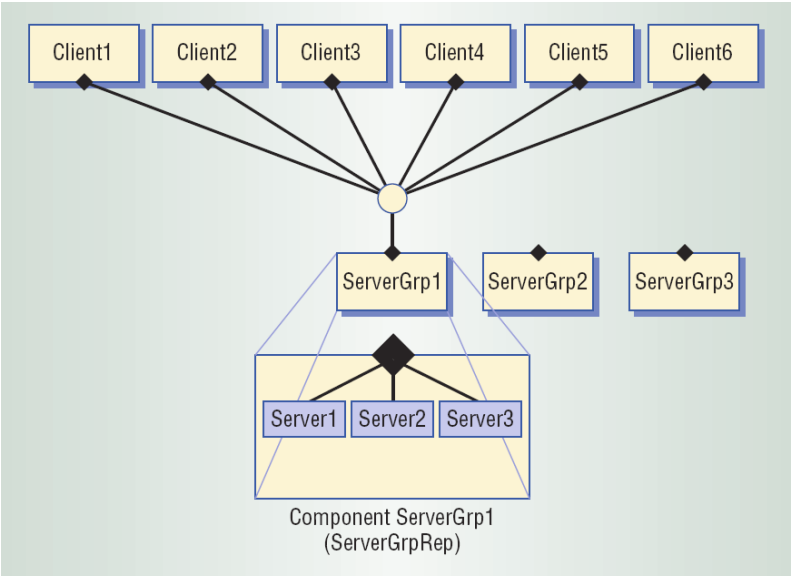


Figure 6.2: Architecture model of the client-server system

The ServerGroupT.addServer() operator finds and adds an available ServerT component to a ServerGroupT instance to increase its capacity. The ClientT.move(ServerGroupT toGp) operator disconnects a ClientT component from its current ServerGroupT instance and reconnects it to a new ServerGroupT instance, *toGrp*. Adaptation actions target client-side latency and allow (1) addition of servers to a group to offset and balance load, and (2) moving of a client to a different server group to circumvent connection problems. Prior to the development of Stitch, these actions were provided directly by the architectural operators.

Associated with each client, an invariant checks to see if the perceived response time is less than a predefined maximum response time (line 1 below, self refers to a client instance). If the invariant fails, an adaptation strategy is invoked to choose one of two adaptation actions, shown below in a pre-Stitch representation. This strategy first checks to see if the current server group's load exceeds a threshold value. If so, it adds a server to the group to decrease the load and thus decrease response time. However, if the available bandwidth, as reported by Remos probes, between the client and the current server group drops too low, the strategy moves the client to another group to get higher available bandwidth and lower response time.

```
1  invariant self.responseTime < maxResponseTime !-> responseTimeStrategy(self);
2
3  strategy responseTimeStrategy (ClientT c) {
4      let g = findConnectedServerGroup(c);
5      if (query("load", g) > maxServerLoad) {
6          g.addServer();
7          return true;
8      }
9      let conn = findConnector(c, g);
10     if (query("bandwidth", conn) < minBandwidth) {
11         let g = findBestServerGroup(c);
12         c.move(g);
13         return true;
14     }
15     return false;
16 }
```

To demonstrate the effectiveness of the framework self-adaptation cycle, we conducted an experiment on a dedicated testbed consisting of five routers and eleven machines communicating over 10-Mbps lines. With the testbed, we could vary bandwidth availability on specific network links to induce move-client adaptations. For each 30-minute experiment, the bandwidth was untouched for the first two minutes, then dropped between two clients and their current server group for eight minutes. To induce add-server adaptations, we controlled the clients to send a low number requests for the first ten minutes, then raised the frequency to twice a second for the next ten minutes to generate excessive load on the existing servers. The control variable is whether adaptation is enabled. More details of the experimental setup, simulated load, and data can be found in [CGS+02b].

Using the simulated network bandwidth setting and stress load, we conducted experiments of adaptation on this client-server system, which required a client-experienced latency of less than two seconds. The results showed that, for this application and the specific loads used in the experiment, self-adaptation significantly improved system performance. Figure 6.3 shows experiment results for system performance with and without adaptation. Part (a) shows that, without adaptation, once the latency experienced by each client rose above 2 seconds, it never recovered again to fall below this threshold. On the other hand, part (b) shows that if Rainbow adaptation is enabled, client latencies were restoredto normal levels within about a minute of exceeding the threshold. The latency peaks at the 2-minute mark reflect the drop in bandwidth availability, while the peaks between the 10- and 20-minute marks reflect the high client-request frequency during that 10-minute period.

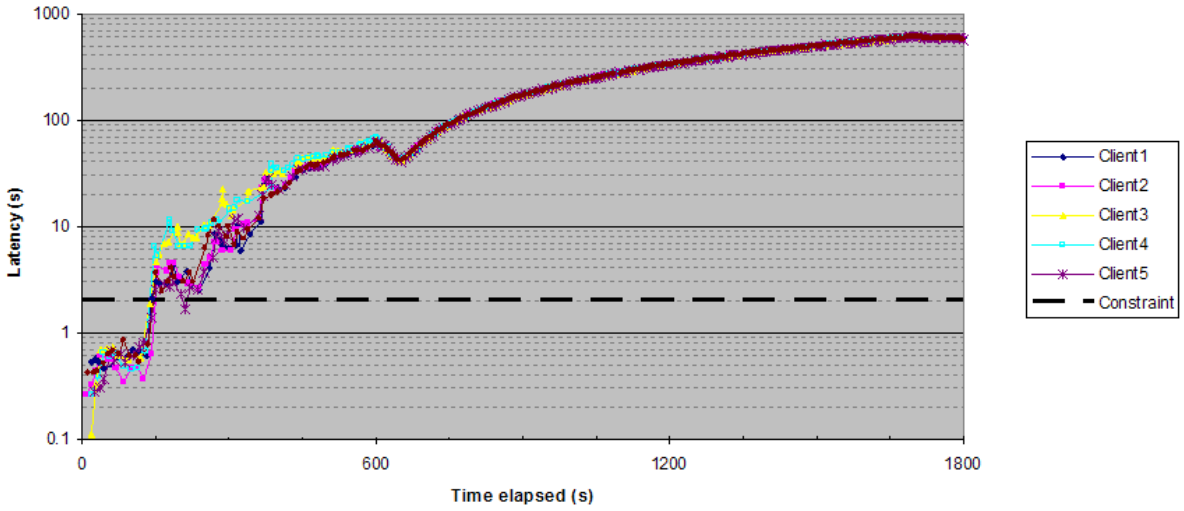
Our experiment also revealed that external repair has an associated latency. In particular, Rainbow took several seconds to notice a performance problem and several more seconds to fix it, indicating that the software architecture-based approach might best suit adaptations that operate on a system-wide scale and fix longer-term system behavior trends. In summary, as evidence toward **V1**, this example application of Rainbow demonstrated an effective roundtrip adaptation cycle using a client-server-style system to address a single quality dimension, *performance*.

6.2 Libra Videoconferencing System

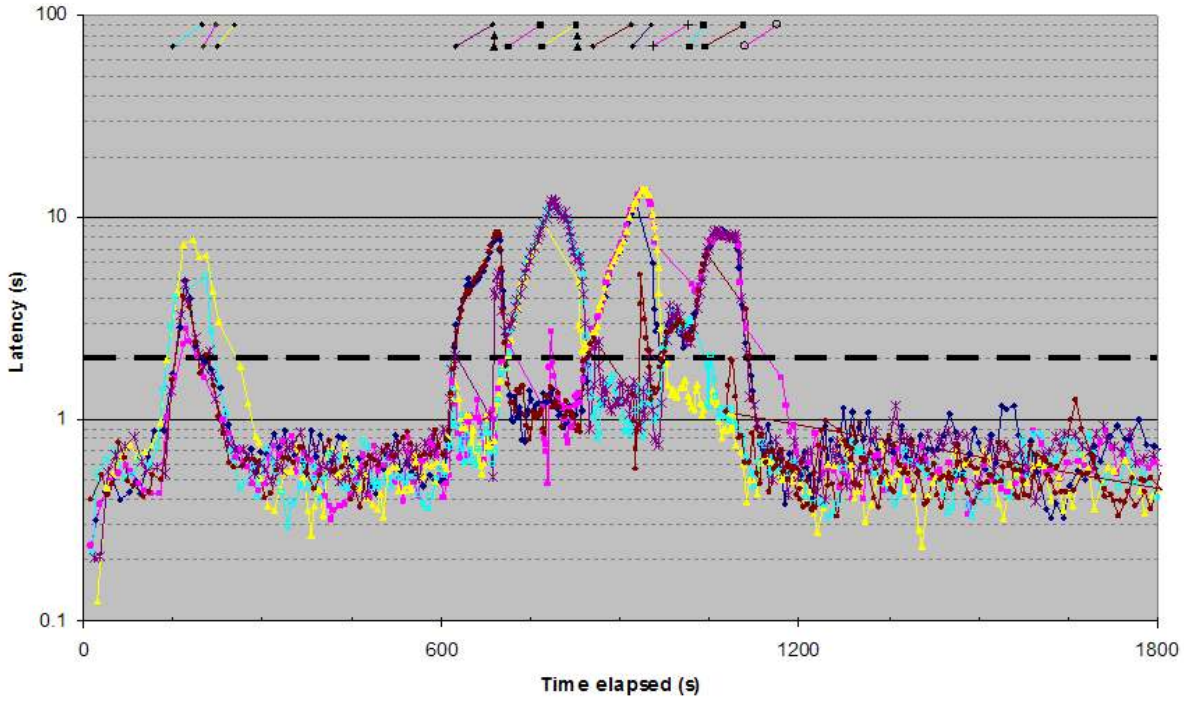
The second Rainbow prototype used a videoconferencing system to explore the coordination of adaptation control between two different, interacting quality dimensions of *performance* and *cost*. This instantiation also improved the separation of control between Rainbow's architectural layer and the target-system layer, added capability of environmental resource measurement and discovery, and provided insight into what constituted shared infrastructures of adaptation.

Specifically, this prototype explored two issues: (1) adaptation for multiple concerns via two interacting adaptation managers, and (2) a more generic scheme for the translation infrastructure. The translation infrastructure consisted of a set of translators that interacted via a repository containing four kinds of mappings: element types, element instances, operators, and exceptions. Three kinds of translators reconciled information from system to model, from model to system, and between models of the adaptation managers. Details can be found in [CHG+04, GCH+04].

The dynamically composed system consists of a videoconferencing session with five participating users. Two of the participants run the Vic/SDR videoconferencing tools, which use the



(a) Control run, adapation disabled



(b) Adaptation run; ticks at the top mark the start and end of adaptation processes per client

Figure 6.3: Experiment result, with and without Rainbow adaptation

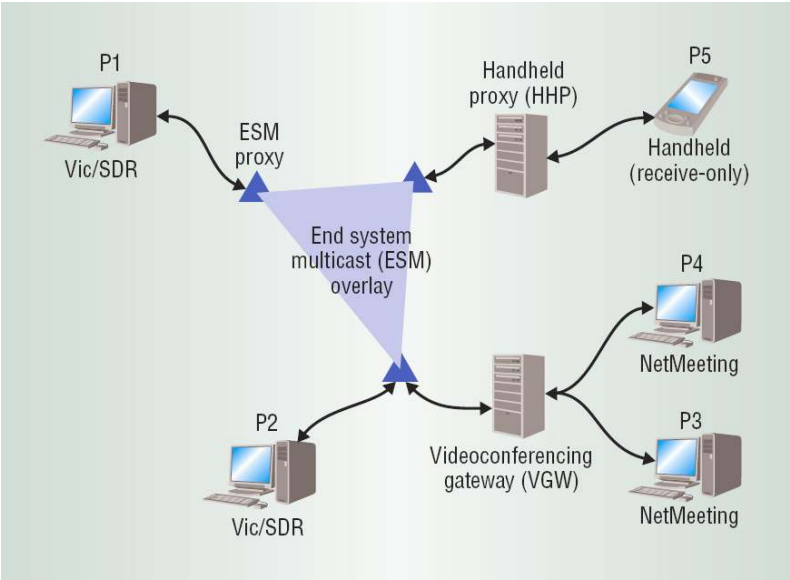


Figure 6.4: Architecture model of the Libra videoconferencing system

Session Initiation Protocol (SIP) and IP multicast. Two other participants run NetMeeting, which uses the H.323 protocol and unicast. The final participant uses a handheld device, which runs a slightly modified version of Vic. Since the handheld device cannot perform protocol negotiation, a handheld proxy (HHP) joins the conferencing session on behalf of the handheld user. A videoconferencing gateway (VGW) that supports both H.323 and SIP translates the protocols for NetMeeting and Vic users. Finally, to allow efficient communication among all participants across wide-area networks, the system uses Narada, an end-system multicast overlay consisting of three proxies, to provide the multicast functionality.

The service provider cares simultaneously about ensuring the delivery of smooth and good quality video to the users (performance) while keeping the cost of providing the videoconferencing service low. For example, if only one NetMeeting user remains online, the system should switch to a low-cost gateway. As another example, the system should ensure the delivery of smooth and good quality video to the users. Performance analysis uses service load metrics and simple connectivity analysis of topology. Cost analysis uses a simple weighted sum of the costs of individual services within the composed system. These analyses identify process load, usage cost, and connectivity as the properties to monitor.

Because this system has a peer-to-peer nature with service provided by various components, we define a service-coalition style for the videoconferencing system based on stakeholder concerns. The architecture model, as illustrated in Figure 6.4, shows the service-coalition-style system with five video application components, two video service components, and one complex connector for the end-system multicast overlay. The major elements of the style include:

- Types: VicT, NetMeetingT, HandheldT, GatewayT, HHProxyT, ESMProxyT, ConnectionT
- Properties: GatewayT.cost, GatewayT.load, ConnectionT.bandwidth
- Operators: HandheldT.move(HHProxyT toHHP), NetMeetingT.move(GatewayT toVGW)

The HandheldT.move(HHProxyT toHHP) operator switches the handheld user to a new handheld proxy, while the NetMeetingT.move(GatewayT toVGW) operator switches the NetMeeting user to a new video gateway. The adaptation strategies, shown below in pre-Stitch representation, enable dynamic user joining and leaving, replacement of service components to eliminate video-quality problems, and reconfiguration of the system to reduce cost or improve performance.

```
1  invariant bandwidthToHHP(self) > minHHBandwidth) !-> HHBandwidthStrategy(self);
2  invariant self.cost / numberOfNMusers < maxVGWUnitCost !-> VGWCostStrategy(self);
3
4  strategy HHBandwidthStrategy (HandheldT hh) {
5      let hhp1 = findBestHHP(hh);
6      hh.move(hhp1);
7      return true;
8  }
9
10 strategy VGWCostStrategy (GatewayT vgw) {
11     let vgw1 = findLowestCost(GatewayT, "load"); /* lowest cost given cur load */
12     if ((query("cost", vgw1) / numberofNMusers) < maxVGWUnitCost) {
13         let users = {select u : NetMeetingT in sys.components | connected(u,vgw)};
14         foreach (NetMeetingT u : users) {
15             u.move(vgw1);
16         }
17         return true;
18     }
19     return false;
20 }
```

The first invariant (line 1 above) is associated with components of type HandheldT, while the second invariant (line 2) is associated with components of type GatewayT. At runtime, when either invariant is violated, the adaptation engine executes the corresponding adaptation strategy. For example, when the available bandwidth between the handheld user and the handheld proxy drops too low, the engine moves the handheld user to a better handheld proxy. When the unit cost of the gateway VGW becomes too high—such as when a NetMeeting user leaves the session—the engine switches the NetMeeting users connected to VGW to the lowest-cost gateway that can handle the load. A conflict between concerns of performance and cost is possible, such as when a performance-targeted adaptation pushes the system's total service cost above a maximum threshold. Although not addressed in this prototype, as we present in another instantiation, a composite utility function can help resolve such conflicts.

In summary, as evidence toward **V1** and **V5**, this example application of Rainbow demonstrated a second instantiation of the adaptation framework on a different style of system, service-coalition, addressing two interacting quality dimensions of *performance* and *cost* (but without considering automated trade-off analysis), reusing adaptation infrastructures from prototype one (100 of 102 kLoC, or ~98%), and using a more generalized infrastructure for translation between model and system.

6.3 University Grade System—Security Domain

A third application of Rainbow explored how self-adaptation could be applied to the *security* domain, with particular attention to two issues: (1) how to model a quality concern that is not immediately quantifiable, and (2) the potential interplay of adaptations between two attributes

like security and performance that exhibit fundamental trade-offs. Although this example was not actually instantiated for experimentation, it did achieve the following:

1. Successfully represented security concerns (*risk*) in an architectural style;
2. Identified security-targeted plug-in elements for a Rainbow instantiation; and
3. Provided an example of adaptation strategies that required trade-off.

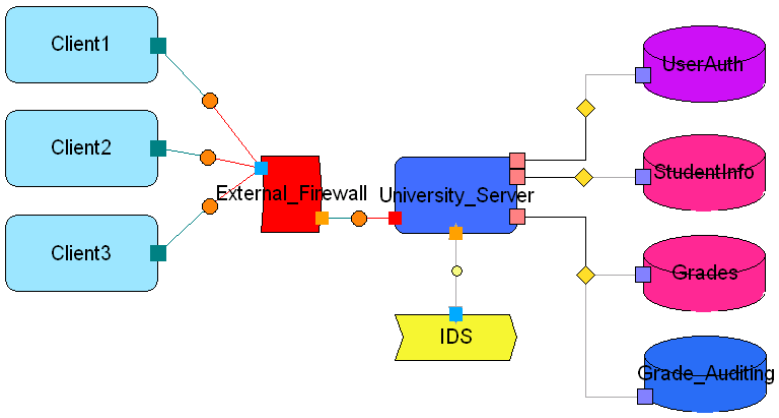


Figure 6.5: Architecture model of the University Grade System

This Rainbow application exemplifies a typical university system that aims to provide timely and ubiquitous student information access. Such a system might consist of a server behind a firewall, connected to one or more backend databases, serving one or more clients. To protect the data assets, the university partitions the student information into three databases, one that stores user account information for authentication, a second that stores student personal information, and a third that stores student grades. Partitioning the data in this fashion is one security engineering approach to prevent the compromise of one database from exposing all data.

In our scenario, a student might attempt to hack into the system to change grades, and the university wants to prevent, or at least be able to detect, such an intrusion.[1] At the same time, the university wants the system almost always to be able to fulfill student requests within a reasonable amount of time. When a suspected intrusion occurs, a security engineer may react with one of a number of escalating responses: turn on auditing, switch authentication scheme, sandbox, move grades data, close off connections, partition network, and turn off services. Note that each of these countermeasures has some negative impact on the responsiveness of the system to student requests, from short delays to zero response.

For security analysis, Rainbow requires a way to quantify security risk so that it can automatically determine when a security adaptation is needed and when one has succeeded. To that end, Butler's Security Attribute Evaluation Method (SAEM) [But02] allows one to analyze and determine an overall risk index for a system. First, one identifies a set of undesirable outcomes, asset components, vulnerabilities in the system, threats to the system, and potential countermeasures against each threat. Then one develops threat models that estimate the threat probability

[1] N.B.: *Despite careful intrusion detection, Easter Eggs inevitably surface in the most serious of documents, to the author's chagrin. If you find any, please do not return to its owner, but keep for you own satisfaction.*

```
System UniversityGradeSystem : SecureClientServerFam = new SecureClientServerFam
    extended with {
    Property overallRisk : ProbabilityT = 0.1;
    Property riskThreshold : ProbabilityT = 0.45;
    Property undesirableOutcomes : OutcomeListT = < "productivity loss", "resource
        loss", "negative reputation" >;
    Property worstDamages : DamageVectorT = <
        [ value : float = 250.0; unit : string = "$" ],
        [ value : float = 5000.0; unit : string = "$" ],
        [ value : float = 15000000.0; unit : string = "$" ]
    >;
    Component Client1 : PrincipledClientT =  new PrincipledClientT extended with {
        Property experRespTime : float = 0.0;
        ...
        rule performanceConstraint = invariant self.experRespTime <= MAX_RESPTIME;
    };
    Component University_Server : SecureServerT = new SecureServerT extended with {
        Property load : float = 0.0;
        Property intrusionProb : ProbabilityT = 0.0;
        Port receiveRequest : SecureServerPortT = new SecureServerPortT;
        Port useInfo : SecureDataUsePortT = new SecureDataUsePortT;
        Port scan : RequirePortT = new RequirePortT;
        Port useGrade : SecureDataUsePortT = new SecureDataUsePortTb1;
        Port useDir : SecureDataUsePortT = new SecureDataUsePortT;
        ...
        rule securityConstraint = invariant self.intrusionProb <= MAX_IDPROB;
    };
    Component StudentInfo : SecureDbT =  new SecureDbT extended with {
        ...
        Property threatModel : ThreatModelT = {
          [ threat : ThreatT = "unauthorized access";
            asset : AssetT = "ssn";
            threatProb : ProbabilityT = 0.1;
            damagePotential : DamageVectorT = <
                [ value : float = 40; unit : string = "%" ],
                [ value : float = 10; unit : string = "%" ],
                [ value : float = 90; unit : string = "%" ] > ],
          [ threat : ThreatT = "virus";
            asset : AssetT = "ssn";
            threatProb : ProbabilityT = 0.25;
            damagePotential : DamageVectorT = <
                [ value : float = 10; unit : string = "%" ],
                [ value : float = 20; unit : string = "%" ],
                [ value : float = 20; unit : string = "%" ] > ]};
        Property effectProfile : EffectProfileT = {
          [ threat : ThreatT = "unauthorized access";
            mechanism : CapabilityT = "plaintext password";
            effectiveness : ProbabilityT = 0.01 ],
          [ threat : ThreatT = "virus";
            mechanism : CapabilityT = "virus scanner";
            effectiveness : ProbabilityT = 0.9 ]};
    };
    ...
};
```

Figure 6.6: Excerpted Acme description of the University Grade System

per asset and its damages potential in case of compromise, along with effect profiles that relate each threat with a countermeasure and its probabilities of effectiveness. Figure 6.6 illustrates how these security concerns are captured for the university system.

The property overallRisk records the security risk value computed using the SAEM; risk-Threshold specifies the maximum risk value tolerated for this system; undesirableOutcomes lists the outcomes of security compromise that the system owner would like to prevent; and worstDamages estimates the maximum dollar values that would be lost for corresponding outcomes. An example threat model and effect profile are shown for the StudentInfo database component, indicating that "ssn" is an asset susceptible to the threat "unauthorized access," which has a 10% likelihood of occurring and a damage potential indicated as percentages of the worst damages. An effect-profile entry indicates that "plaintext password" has a 1% effectiveness against "unauthorized access." The UniversityServer component shows a property that captures the probability of an intrusion occurring on the server. The intrusion-probability value would be updated by a gauge that interprets the readings of a component-based intrusion-detection probe, which scans port activities on the server to determine whether an intrusion might be taking place.

Using the threat models and estimated intrusion probability, the SAEM provides a means to determine a relative risk index, from which one can compute an overall risk score for a system configuration. The security analysis thus identifies intrusion probability as a property to monitor. As with prior systems, performance analysis identifies process load and connectivity as properties to monitor. From the system, security, and performance analyses, we define a composite client-server and shared-data style with explicit performance and security properties—secure-client-server—for the university grade system. The architecture model, illustrated in Figure 6.5, represents the secure-client-server-style system with three clients connected to the university server through a firewall, accessing three databases. Two security components are shown: an IDS (intrusion detection system) and a Grade Auditing database. Built into this style are security-related types and attributes, as illustrated in part by the Acme description of the system in Figure 6.6, which allow the security risk score to be computed. The style also includes:

- Types: SecureServerT, PrincipledClientT, SecureDbT, IDST, SshConnT, HttpConnT
- Properties: SecureServerT.intrusionProb, SecureServerT.load, HttpConnT.bandwidth
- Operators: SecureServerT.replicate(int delta), SecureServerT.attachIDS(IDST ids), SecureDbT.audit(boolean on), PrincipledClientT.isolate(), ...

The performance-targeted SecureServerT.replicate(int delta) operator increases or decreases the replication count of the server by some delta. The remainder are security-oriented operators, and note that the list highlights but a subset of possible operators to counter intrusion. The SecureServerT.attachIDS(IDST ids) operator attaches an intrusion detection system to the server to monitor it for potential intrusion attempts; the SecureDbT.audit(boolean on) operator turns database audit trail on or off; and the PrincipledClientT.isolate() operator temporarily blocks a client associated with an authentication principle from accessing the system.

Adaptation strategies, represented in Stitch in Figure 6.7, either increase the server replication count when the client-experienced response time exceeds a maximum threshold, or counter intrusion when the intrusion probability hits a high threshold. While not shown, strategies that achieve the reverse can also be specified. The architectural constraints and strategies have been specified using the new Rainbow framework conventions. The constraints to trigger adaptation

```
1  module univ.grade.strategies;
2  import model "UniversityGradeSystem.acme" { UniversityGradeSystem as M,
       SecureClientServerFam as T };
3  import op "tool.Util" { Util as U };
4
5  define boolean respTimeVio = exists c : T.PrincipledClientT in M.components
6      | c.experRespTime > M.MAX_RESPTIME;
7  define boolean intrudeVio = exists s : T.SecureServerT in M.components
8      | s.intrusionProb > M.MAX_IDPROB;
9
10 strategy FixResponseTime [ respTimeVio ] {
11   t0: (true) -> tacticReplicateServer() | done;
12 }
13
14 strategy CounterIntrusion [ intrudeVio ] {
15   define boolean intrudeClient = exists c : T.PrincipledClientT in M.components
16       | U.computeIntruderProb(c.principle) > M.MAX_IDPROB;
17   define boolean intrudeDb = exists db : T.SecureDbT in M.components
18       | db.intrusionProb > M.MAX_IDPROB;
19
20   t1: (intrudeClient) -> tacticIsolateClient() | done;
21   t2: (intrudeDb) -> tacticAuditDatabase() | done;
22   t3: (default) -> tacticAttachIDS() | done;
23 }
```

Figure 6.7: Example adaptation strategies for the university grade system

are specified in the architecture description of the system, shown in Figure 6.6 on line 13 for performance and line 24 for security. In most circumstances, a performance constraint violation results in the FixResponseTime strategy being chosen, while a security constraint violation results in the CounterIntrusion strategy being chosen.

However, since CounterIntrusion strains server load, when the server load is already high, CounterIntrusion will score lower and FixResponseTime will be selected. Then, in the next adaptation cycle, the CounterIntrusion strategy is selected to address the security issue. While such adaptation selection may not be desirable if security violations preempt performance ones, it shows that quality trade-offs depends on the present system conditions for strategy selection.

In summary, as evidence toward **V1** and **V5**, this example demonstrated the application of Rainbow to a composite style of system, client-server with shared-data, addressing primarily the non-trivial quality dimension of *security* while also highlighting trade-off against *performance*.

6.4 TalkShoe

An important validation of the Rainbow approach is to demonstrate its usefulness in a real-world system. Ideally, we would like to have different IT organizations adopt the Rainbow adaptation technique. However, for the purpose of the thesis, it suffices to show one instance: apply Rainbow to the IT infrastructure of one company for certain management objectives. TalkShoe served this purpose. Most importantly, the TalkShoe instantiation of Rainbow showed the successful use of the framework by someone other than its author and provided data on customization effort as supporting evidence for the *cost-effectiveness* of the approach. Furthermore, TalkShoe offered us some lessons for improving the Rainbow approach for future use.

6.4.1 Background

TalkShoe is a new technology company in Pittsburgh that offers web-based voice talk shows and discussion groups, known as *talkcasts*, to its customers. Customers can create talkcasts on their favorite topics, then host episodes and invite others to join the discussion. The primary application interface is a TalkShoe Java client running on the customer's computer. It is required to host a talkcast *episode*. Customers can join a talkcast via a Java client or a regular telephone, like a conference call. Business users can use TalkShoe to arrange various kinds of virtual, internal meetings. Finally, the host of a talkcast episode can choose to archive the episode at its termination, and the system automatically exports the recording to an MP3 file. MP3 files comprise important tangible assets for TalkShoe customers.

Like many contemporary webservice companies, TalkShoe's web service infrastructure integrates open source, as well as third-party proprietary software, making it highly difficult to determine problem causes, and nearly impossible to eliminate all transient, system integration bugs. More importantly, most transient problems are not detected until they manifest as faulty, customer-visible system behavior. TalkShoe engineers hoped that Rainbow could at least provide them the advantage of early detection.

In early-November 2006, Carl Paradis helped to arrange an audience with the TalkShoe engineers. I sent an initial case study proposal through Carl, and negotiations ensued informally to determine a fit for the case study. In early-February 2007, Chief Architect Robert (Bob) Pawlowski arranged an initial meeting with me to see a research presentation on Rainbow and assess collaboration potentials. Bob discussed some of the difficulties they encountered managing their server infrastructure, and saw potential applications of the Rainbow technique to close the loop of control. From this initial meeting, we established an understanding that

1. I would have access to their system, including the pre-production environment (and possibly even have my own local setup except for the third-party conference bridge).
2. Bob would be willing to work with me on the domain-specific code.
3. I would work closely with Bob to customize Rainbow for TalkShoe, which means, in addition to providing Rainbow software support, I would create the architecture model and write an initial Stitch adaptation script. Bob would provide me with a description for adaptation, and he would be expected to modify the script on his own later.

6.4.2 TalkShoe Infrastructure

The TalkShoe infrastructure shares commonalities with the Libra videoconferencing system, in which a hosting user sets up a teleconference and participants join or leave it after the conference starts. Unlike Libra, a session consists only of one type of application client, there is no video, and voice delivery is done via the Internet and a specialized teleconference bridge. Figure 6.8 shows a simplified architecture of TalkShoe's infrastructure. The TalkShoe Server is a custom standalone server that uses Java NIO to connect to many clients. It serves one or more TalkShow Hosts and regular TalkShoe Live Clients (not shown in diagram). During talkcast sessions, each TalkShoe Live Client connects to the TalkShoe Server through Java NIO using proprietary binary protocol. TalkShoe also offers a web-based client that first connects via an HTTP server, then

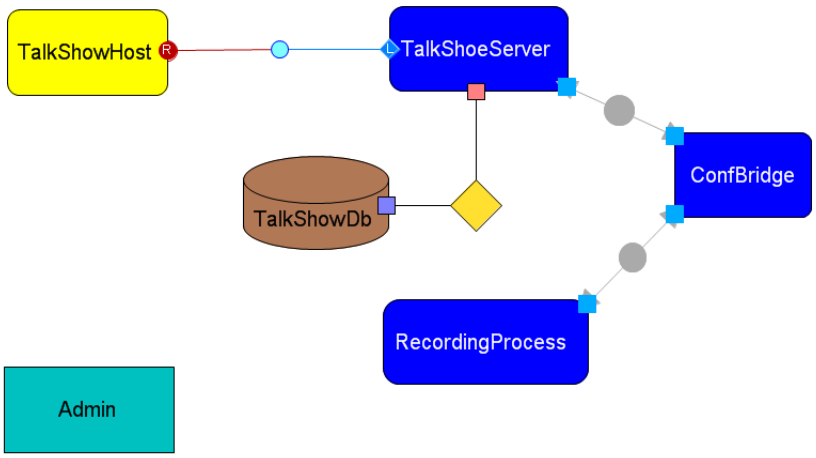


Figure 6.8: A simplified architecture model of TalkShoe's infrastructure

communicates with the TalkShoe Server using the Jabber protocol.

Data for TalkShow episodes are stored in the TalkShow Database. Some users join talkcasts by teleconference using the Conference Bridge component (users not shown). At the command of the TalkShow Host, the Conference Bridge may output the audio content of an episode to file, and a Recording Process component monitors the Conference Bridge for audio files to export to MP3. The box labeled Admin is not actually part of the architecture, but captures a sys-admin who gets notified when a problem occurs. As explained later, Rainbow uses properties on this Admin component to track processed notifications.

6.4.3 Problem Scenarios for Adaptation

In the February 2007 meeting, Bob identified four problem scenarios where Rainbow can help:

1. Web-based talk-show setup: the Tomcast clusters comprising the TalkShoeServer transiently exhibits long service time, so Rainbow could be used for performance adaptation of the Tomcat configurations.
2. Conference drop: once or twice a week, the ConfBridge drops out and requires restart, so Rainbow could be used to detect this dropout and automatically issue a restart.
3. "Talk" status on client console stops updating: the TalkShow client console shows a "talk" status for each participant of the episode, but occasionally, the talk status stops updating, so Rainbow could potentially be used to determine the problem origin in the infrastructure.
4. Management of episode recordings: occasionally, something happens in the audio recording pipeline that prevents the RecordingProcess from producing MP3 files, so Rainbow could be used to notify the sys-admin of missing audio files before customers discover the problem. One complicating factor is that the audio recording process takes time to complete, so it's unclear when an unavailable audio file is actually missing. On the other hand, the sys-admins know from experience that the recording ought to take no more than 2-4 hours, and customers usually complain after about 3-4 hours.

Based on a preliminary assessment, Table 6.1 characterizes the four scenarios in terms of TalkShoe's priority (P), relevance to Rainbow (A), and estimated level of effort to customize Rainbow to tackle the scenario (E). Scenario #2 appears to be the best candidate for an initial Rainbow-TalkShoe effort because it involves at least three components in the TalkShoe infrastructure, has existing effector actions that administrators currently perform manually, and may require a couple of tactics and one or more strategies. However, it was unclear what properties could be used to check for problem to adapt, and this scenario boils down to only one objective: continue operation of, and minimize disruption to, existing teleconferences.

Table 6.1: TalkShoe scenario feasibility assessment

	P	A	E
1	low	high	med
*2	high	med	low
3	med	low	high
4	high	med	med

Scenario #4 is a likely second candidate, whereas #1 may not be worthwhile, and #3 amounts to distributed system debugging, not fit for Rainbow's purpose. After discussing scenario feasibility, Bob understood more clearly that Rainbow is most useful for monitoring day-to-day operations and responding to undesirable conditions. Bob soon recognized a better TalkShoe scenario for Rainbow: adapt a talkshow to prevent more users from joining once Rainbow detects a rise in latency for existing users. Unfortunately, this scenario seemed overly complicated, ambitious, and time-consuming, so we decided to save it as a potential future pursuit. Apparent from similar discussions with TalkShoe engineers, once an engineer understands the potential of architecture-based adaptation, s/he can quickly find applicable scenarios.

6.4.3.1 Scenario 4: Ensuring Presence of Episode Recording

In order to get TalkShoe engineers on board and to recognize the benefits of Rainbow, Bob and I decided to pursue scenario #4, which TalkShoe deemed to be high-priority and required manageable effort to implement. In this scenario, TalkShoe users were able to keep archives of their own TalkShoe recording. A short pipeline process began from the point the user clicked "Stop" to terminate the talkshow, to the point a transcoded MP3 recording of the show appeared on the user's TalkShow homepage (while an RSS feed was simultaneously generated). These MP3 files were the only "tangible" assets to TalkShoe customers. The problem was that, for some reason as yet undetermined, an MP3 file was not always successfully produced. When the system failed to generate the necessary MP3 files, users expecting them became quite upset. This scenario extended to a service that allowed a user to upload her own audio files, which the TalkShoe system then processed before putting the MP3 into the database and updating the user's TalkShoe homepage. Here, the biggest point of failure seemed to be the upload step, especially since the audio files were generally quite large.

At a high-level, the observable quality attribute was the *availability of talkshow asset to the user*. To detect the presence of a problem, four probes were potentially needed for the following

purposes, and at least the first and the last must be present for meaningful problem identification.

1. Detect the talkshow episode "stop" operation from the user
2. Detect the presence of a wave file and the status of its transcoding into MP3
3. Detect the upload state of an audio file from a user
4. Detect an entry in the database as a sign of successful MP3 production

Once we could detect the episode-stop event and the MP3 database entry, we wanted to implement a simple strategy: if *no* MP3 recording was found in the database for an episode that had been stopped for longer than a predefined observation window, say one hour, then notify the sysadmin of a possible recording problem. The email notification action was the *only* adaptation effector. Perhaps a more substantive adaptation was possible, but the engineers had not determined one for this scenario yet. In the architecture model, information about missing MP3 files would logically reside as a property of the TalkShowDb component. More specifically, we could use a TalkShowDb property *episodeAbsentRecording*, updated by a gauge, to track the status of a set of episodes that were expected to have an MP3 recording eventually.

6.4.3.2 Adaptation Workflow for Scenario 4

Let us walk through the customization items needed for this scenario. For this scenario, there was effectively one objective—the availability of the talkshow recording—so we did not need to worry about trading off across utility dimensions.

1. First, we needed to develop two probes: the **EndRecordingProbe** to detect the episode-stop event from the user, and the **CheckForRecordingProbe** to query the database for an MP3 file. There was one small complication: CheckForRecordingProbe required knowledge of the unique TalkShow Episode identifier (*epID*) associated with the episode that was stopped, which was known only to the EndRecordingProbe. However, these two probes resided on different machines, and no communication path existed between Rainbow probes across machines. So, we needed a way to connect these two pieces of data.
2. Next we needed a gauge to update the model property TalkShowDb.*episodeAbsentRecording*. On initial configuration, the gauge would be supplied with an observation time window, so that it knew when to update the set of *episodeAbsentRecording*. This gauge would need to have knowledge of the epIDs for the stopped episodes, as well as the status of MP3 recordings in the database. The two probes could provide these two pieces of data. Furthermore, since the RecordingMonitorGauge served as a single location to track both pieces of data, it could also coordinate this information between the two probes. Thus, we had the following sequence of events:
   (a) When the EndRecordingProbe detected an episode-stop event, it passed this information to the RecordingMonitorGauge via the Probe Bus.
   (b) RecordingMonitorGauge forwarded this information to the CheckForRecordingProbe via a Probe-Configure operation.
   (c) CheckForRecordingProbe queried the database periodically and published the status of the expected MP3 file to the gauge. The status could be one of

URL_NOT_FOUND or RECORDING_FOUND.

(d) The RecordingMonitorGauge kept time on all the epIDs that it was tracking, and published an updated set of *episodesAbsentRecording* to the architecture model whenever a timer exceeded the observation time window or a recording status changed to RECORDING_FOUND.

3. In the architecture model, we needed to define constraints to trigger notification whenever the set of *episodeAbsentRecording* changed. Since TalkShoe engineers did not want redundant notifications, the constraint needed to be able to distinguish between *new* episodes that were absent recording and episodes of which the sys-admin had already been notified. For this purpose, the Admin component external to the architecture in Figure 6.8 served to track those episodes for which a notification had been sent to the administrator.
4. To respond to the constraint trigger, we needed a Stitch script. Since this scenario required only a notification action, the script would simply contain one strategy, NotifyOnExpire, that invoked one tactic, notifyByEmail, to send an email notification for the new epIDs.
5. We needed an effector to send the email, which notifyByEmail would eventually invoke.

6.4.3.3 Development and Deployment

After a couple of brainstorming sessions, we were ready to start implementing the two probes and the notification effector. I packaged together the Rainbow dependent Jar files, a set of Rainbow customization files, a sample set of probes and gauges, and the Rainbow APIs pertaining to probes, gauges, and effectors. This comprised the *software development kit* (SDK) for Rainbow, in the form of an Eclipse workspace plug-in, which Bob would use for the customization to TalkShoe. In terms of development interactions, I met with Bob at TalkShoe twice a week to work on customizations that required his domain knowledge and the TalkShoe environment. Individually, I worked on the architecture model and the Stitch script (counted in the time data), as well as feature addition, bug fixes, and general enhancements to Rainbow (not counted).

In the first session (see Table 6.2), Bob created stubs for the EndRecordingProbe and the CheckForRecordingProbe to (a) gain familiarity with Rainbow's APIs, (b) simulate episode-stop event and database lookup outside of the TalkShoe environment, and (c) provide me the parts to connect other pieces of Rainbow. Based on my cursory understanding of the TalkShoe infrastructure, I developed the Acme family and system instance that comprised the architecture model of TalkShoe. I also enhanced the Rainbow framework to support gauge configuration of probes. In the second session, I implemented the RecordingMonitorGauge to coordinate the two probes and populate the architecture model, and Bob wrote an EmailNotifyEffector using Java mail to send emails. I then composed the notifyByEmail tactic and NotifyOnExpire strategy for the notification adaptation, shown in Figure 6.9. Bob and I hooked up the model, the Stitch script, and the effector during session 3, refined the probes to generate useful data in session 4, and tested the adaptation roundtrip on Bob's development machine during sessions 5–7. Deployment into the pre-production environment would mark the first exercise of all the cross-machine gauge and probe communication code, and that succeeded on session 10.

Bob decided that once Rainbow passed pre-production testing, he planned to deploy it with the July 2007 production release. However, in the end, the Rainbow code was not deployed

into production, in part due to Bob's schedule constraint, and in part because a later TalkShoe software version appeared to have solved the episode-recording issue, eliminating Rainbow from TalkShoe's set of critical features for production release.

```
module talkshoe.tactics;
import model "TalkShoeSys.acme" { TalkShoeSys as M, TalkShoeFam as T };
import op "talkshoe.operator.ArchOperator" { ArchOperator as S };
import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util

tactic notifyByEmail (set{T.TalkShoeDataT} dbs) {
  set unnotifiedSet = {};
  condition {
    // some episodes are not in Admin's notified set, i.e., not notified
    exists db : T.TalkShoeDataT in dbs
        | Set.size(Set.diff(db.episodesAbsentRecording,
                            M.Admin.episodesNotified)) > 0;
  }
  action {
    for (T.TalkShoeDataT db : dbs) {
      unnotifiedSet = Set.union(unnotifiedSet,
          Set.diff(db.episodesAbsentRecording, M.Admin.episodesNotified));
    }
    S.emailNotify("localhost", unnotifiedSet);
    set notifiedSet = Set.union(M.Admin.episodesNotified, unnotifiedSet);
    Model.setModelProperty(M.Admin.episodesNotified, notifiedSet);
  }
  effect {
    // those new episodes should now be in Admin's notified set
    Set.size(Set.diff(M.Admin.episodesNotified, unnotifiedSet)) == 0;
  }
}
```

```
module talkshoe.strategies;
import lib "talkshoeTactics.s";

define boolean isTalkShoeStyle = Model.hasTypes(M, "TalkShoeDataT");

// This Strategy simply notifies a developer of list of bad episodes.
strategy NotifyOnExpire
[ isTalkShoeStyle ] {
  set dbs = { select db : T.TalkShoeDataT in M.components
      | U.size(db.episodesAbsentRecording) > 0 };
  boolean hasUnnotified = exists db : T.TalkShoeDataT in dbs
      | Set.size(Set.diff(db.episodesAbsentRecording,
                          M.Admin.episodesNotified)) > 0;

  t0: (hasUnnotified) -> notifyByEmail(dbs) {
    t1: (!hasUnnotified) -> done;
  }
}
```

Figure 6.9: Tactic notifyByEmail and Strategy NotifyOnExpire

6.4.4 Data and Result

While performing this case study, we kept track of the time spent. Total development time for the MP3 scenario lasted almost 34 hours, with the activity breakdown detailed in Table 6.2. In the beginning, Bob and I spent more than 3 hours on one-time development environment setup.

Table 6.2: TalkShoe scenario development activity data

Ses	Date	Time	Development Activity	Dur (hr)
1	2007.04.11	1415–1800	Initial dev env't; Gauge + Probe	3.75
	2007.04.13	1315–1515	Model: architecture style + instance defined	*3.0**
2	2007.04.13	1515–1915	Gauge + Effector	4.0
	2007.04.14	1125–1501	Stitch: script defined + effector plumbing	*3.6*
3	2007.04.18	1000–1245	Hooked with model + Stitch; Effector	2.75
4	2007.04.20	1520–1745	2 Probes + simulator	2.4
5	2007.04.25	1520–1750	Partial roundtrip debug + Stitch script	2.5
	2007.04.25	1525–1640	Model: Port types, Admin component + Stitch	[*1.25*]
	2007.04.26	0130–0230	Model: families and episode record field type	*1.0*
	2007.04.27 2007.04.28	0350–0405 0019–0025	Model: architecture constraint for absent recordings + property type for EpisodeInfo	*0.35*
	2007.04.27	2055–2319	Stitch: script completed, trigger condition added	*2.4*
6	2007.05.02	1012–1130	Roundtrip debug + deployment plan	1.3
7	2007.05.08	1112–1230	Deployment on local server; infrastructure bugs!	1.3
	2007.05.10 2007.05.17	1325–1331 0214–0220	Model: modification + rollback of secondElapsed; Architecture model final version	*0.2*
8	2007.06.01	1050–1144	Pre-production deployment	0.9
9	2007.06.05	1425–1725	Pre-production test #2; Gauge-Probe arch bug	3.0
10	2007.06.08	1100–1230	Successful pre-production deployment	1.5
			Total development time:	**33.95**

*: italicized time indicates one-person effort

[]: bracketed time indicates overlap with another time block

In the end, we sank a little over 5 hours to debug Rainbow deployment in the TalkShoe pre-production environment, an activity that is not expected to require as much effort after the initial curve. Therefore, about **26 hours** can be attributed to architecture modeling and customization of Rainbow to the TalkShoe environment, including the development and testing of probes, gauges, and effectors. Of those 26 hours, almost 12 hours involved only one person and the remaining 14 hours involved both persons. Hence, more accurately figured, the customization effort took 40 man-hours, while the overall scenario took 56 man-hours of effort.

6.4.5 Conversations with the TalkShoe Architect

On December 12, 2007, after concluding the Rainbow customization effort with Bob Pawlowski the chief architect at TalkShoe, I met with Bob to acquire his view on the cost-effectiveness of Rainbow. The format took the form of a prompted conversation:

- (*Initial prompt*) What would you estimate to be the time and effort (or difficulty) of modifying the TalkShoe infrastructure to support similar self-adaptations without Rainbow?
- If adaptation concerns would have been considered upfront when designing the TalkShoe

Infrastructure, what would you say the impact to effort/time would have been?

- What would the time and effort be with Rainbow?
- What would be preventing the adoption of Rainbow today?

  The benefit [of Rainbow] was, the more that is done, the more saving [in effort]. [Without Rainbow,] the initial [design and implementation] effort might take about the same amount of time, but would [couple the design of adaptation concerns] into the [TalkShoe] system. [After the initial efforts,] more [adaptation modifications] can be done in a shorter time, [as there is essentially an] amortization of effort, [assuming our design observed] separation of concerns [and was] generalizable.

  [In designing and implementing the TalkShoe infrastructure, we adopted an] Agile [mentality]: do the simplest that could make things work. So, these adaptation concerns would not have come up early on. However, if we *had* to add the adaptation concerns [initially, adaptation engineering of] just the monitoring and action loop [(e.g., capabilities to probe recording status and notify by email)] would have perhaps **added about 5-10% (~2 man-months)** [to the overall development time]. To cover the basic adaptation concerns [for] most problematic [scenarios identified in Section 6.4.3 would have probably required] **another 2 man-months**.

  [The effort to apply Rainbow] depends on the time to learn Rainbow. If it were a mature product with [an integrated development environment] and deployment mechanism, then [it would] definitely [require] less time, say, a month to learn it, a month to build. Then the amount of time saved would be in terms of maintenance.

  [The trick for commercial adoption of Rainbow would be whether engineers] can install Rainbow from one place to all parts of the system, [manage from] one console, update [Rainbow] automatically, so that the engineers can focus on their domain. [There is a] software development movement toward "convention-over-configuration," for example, Ruby-on-Rail, the Java alternative Grails (Groovy + Spring + Hibernate). [So the] Litmus test [for Rainbow would be]: can the engineer use it and get it working in a day. Also, incrementality of the framework [is important/crucial].

From the conversation with Bob, a number of key phrases offer insight into the cost-effectiveness of Rainbow: the more that is done, the more savings; amortization; separation of concerns; generalizable; four versus two (1+1) months; timed saved in maintenance; integrated development and deployment; let engineers focus on their domain; incrementality. In the next chapter, we discuss these points further when evaluating Rainbow's fulfillment of the thesis requirements.

6.4.6 TalkShoe Summary

In brief, the TalkShoe case study demonstrated the following:

1. Evidence suggests that the Rainbow approach is cost-effective (see Section 7.2)
2. Architecture-based self-adaptation has potential adoption in the "real world"
3. There exists at least one engineer other than Rainbow's author who can use the framework

As evidence toward **V1**, **V3**, and **V4**, this Rainbow instantiation for TalkShoe demonstrated its successful application in an external, commercial setting by another individual. Rainbow was customized for an N-tier-client-server-style system with *availability* concern. It provided data on customization effort as supporting evidence for the *cost-effectiveness* of the approach.

6.5 Znn.com News System

Another, critical validation of the Rainbow approach is to demonstrate its ability to achieve adaptation that trades off across multiple objectives. Besides serving as the running example in this thesis thus far, the Znn.com Rainbow instantiation serves the following purpose:

- It explores the composability of adaptations across three different quality attributes;
- It demonstrates an integrated Rainbow framework with explicitly defined customization points, while showing the ease of customizing the framework parts; and
- It fully illustrates the capabilities of the Stitch self-adaptation language.

In this section, we motivate and present the Znn.com system in detail, describe the experimental scenarios and setup, and present the experimental data and self-adaptation engineering results.

6.5.1 Motivation: *Slashdot Effect*

The motivation for the Znn.com instantiation comes from an Internet phenomenon known as the *Slashdot effect*, where an otherwise low-traffic website, shortly after being featured on *slashdot.org*, gets inundated with visitors for a period of time, anywhere from a few hours to a couple days [Ter04]. Now a common term, it describes a similar phenomenon where a website experiences a sudden, unanticipated rise in requests due to a popular event (or sometimes, anticipated, but not to the scale observed), for example, breaking news or superbowl craze [Wik08d].

To illustrate a few instances, on the morning of August 17, 2006, a break in the JonBenet Ramsey case resulted in a local news website, *rockymountainnews.com*, being swamped by news readers. Browsing the site at 10:18 AM took more than 2 minutes to retrieve a blank page with only the local weather showing in the corner (see also a timed `wget` record in Appendix D.1 on page 191). On September 4, 2006, the shocking death of Steve "Crocodile Hunter" Irwin caused the Australian Broadcasting Corp website to be temporarily shut down; similarly, numerous other news sites groaned to a halt. Of note is that Australian Broadcasting Corp's site resumed a few hours later with a low-bandwidth format to cope with the high traffic [Tai06]. A more problematic, revenue-losing case arises when an e-commerce site is shut down by the *Slashdot*-like *effect*, as exemplified by Wal-Mart's shopping site on Black Friday 2006 (see screenshot in Appendix D.1)[2], which remained inaccessible into the afternoon after the initial morning rush [San06, Sch06].

The Australian Broadcasting Corp illustrates the present coping mechanism of most website administrators to the *Slashdot effect*. When they first notice a sudden rise in visit requests, they make a decision to temporarily shut down the site for manual reconfiguration, where usually

[2]Although, arguable in Wal-Mart's case, the Black Friday traffic could have been anticipated, Wal-Mart may have anticipated an initial burst for its storefront but *not* to the *scale* it experienced for its Web-front.

entails resorting to lower fidelity content. In extreme cases, the site might temporarily shut down or simply post a notice to "visit later." There are a number of disadvantages with this scheme:

- The problem may not be discovered in time;
- The manual process means slower response time;
- The chosen solution may be suboptimal; and
- It may result in a potential loss of confidence, revenue, or future business.

An ideal approach would be to achieve site adaptation automatically, and our aim is to demonstrate such self-adaptation with the Znn.com instantiation. It is worth noting that a few modern systems with sophisticated infrastructures, such as Google Gmail, have equipped themselves with similar but limited adaptation capabilities, for example, to instruct its users to "reload in a few seconds" when it detects a sudden peak or other underlying connectivity issues to its servers.

While the Znn.com example is motivated by *Slashdot effect*, we apply Rainbow to address only the dynamic-adaptation aspect of the problem. Other solutions are possible, such as explicitly blocking links from Slashdot.org [Wik08d], using a thirt-party system like Mirrordot to automatically mirror a website that is featured on Slashdot.org [Ter04], and using Akamai to geographically distribute traffic.

To demonstrate Rainbow's self-adaptation capabilities, we compare the use of self-adaptation versus basic manual configurations using a Znn.com setup. We use the expected utility of the system (as described in Section 4.3.6) to its two groups of stakeholders, the providers and users, as the metrics of comparison. By *effective,* we mean that an approach resulted in a high utility for the system. We show that Rainbow self-adaptation is more effective than basic manual configuration on a system like Znn.com. We further show that Rainbow incurs low overhead in the process, as well as achieves composability. Specifically, we explore the following hypotheses:

- Rainbow self-adaptation is more effective than manual reconfiguration.
- Rainbow incurs low resource overhead (1-2%) on crucial resources like CPU utilization.
- Given multiple objective dimensions, Rainbow chooses the most effective strategy.
- Incremental addition of a new objective dimension incurs low development effort, O(days), and incremental addition of new adaptation strategies (once tested to work) requires, at worst, a quick restart of Rainbow.

We now present the Znn.com system and its Rainbow customization.

6.5.2 Rainbow Customization for Znn.com

The typical infrastructure for a news website like *cnn.com* and *rockymountainnews.com* has a three-tier architecture consisting of a set of application servers that serve contents from backend databases to clients via frontend presentation logic. The Znn.com system imitates such a setup. Architecturally, it is a web-based client-server system that satisfies an N-tier style, as illustrated in Figure 6.10.[3] Znn.com uses a load balancer to balance requests across a pool of replicated servers, the size of which can be manually adjusted to balance server utilization against service

[3] Although a common component, the database increases system configuration complexities without adding value to the illustration of adaptation features, and has thus been excluded from our instantiation.

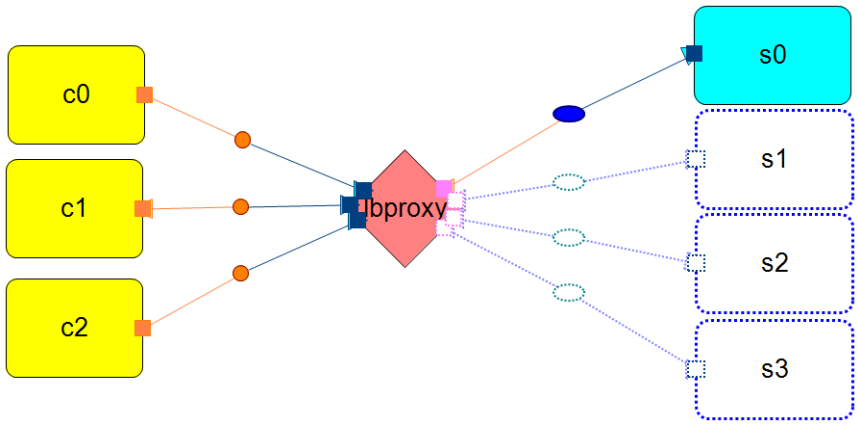


Figure 6.10: Architecture model of the Znn.com system

response time. A set of client processes makes stateless content requests from one of the servers, the servers serve both static files (e.g., images and videos) as well as dynamic contents (e.g., news template populated from periodically updated news source).

Typical of news provider concerns, our quality objectives for Znn.com is to serve news contents to its customers within a reasonable response time range while keeping the cost of the server pool within certain 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 losing customers, we opt to serve minimalist textual contents during such peak times in lieu of providing zero service to the customers. In short, we identify three quality objectives for the self-adaptation of the Znn.com system: (*A*) performance, (*B*) cost, and (*C*) content quality.

Performance analysis outcomes from prior systems with similar performance concerns inform us to monitor the request-response time, server load, and connection bandwidth of the system. Cost analysis identifies the number of active servers as the primary contributor to cost, hence we monitor the server count. For content quality, we characterize different levels of content fidelity ranging from full multimedia to static text, then assign three levels of *high*, *medium*, and *low*. Thus, major elements of the N-tier-client-server architectural style for Znn.com include:

- Types: ClientT, ServerT, ProxyT, HttpConnT
- Properties: ClientT.experRespTime, ServerT.cost/load/fidelity, HttpConnT.bandwidth
- Operators: ServerT.activate(), ServerT.deactivate(), ServerT.setFidelity(int level)

The ServerT.activate() operator activates a ServerT instance, while the ServerT.deactivate() operator deactivates it. The ServerT.setFidelity(int level) operator sets the server content fidelity to the level identified by the input integer. From these operators, we have specified two pairs of tactics with opposing effects. One pair enlists (*1*) or discharges (*2*) servers while the other pair raises (*3*) or lowers (*4*) the server content fidelity. In effect, these tactics allow the service level of the Znn.com system to be stratified into gradients that trade off the various objectives. This enriches the adaptation space over prior examples, which defined only binary modes of adaptation. The following example illustrates how these tactics might interact:

When the response time is high, objective *A* suggests that Znn.com should increment its

server pool size (*1*) if it is within budget; otherwise, Znn.com should switch the servers to textual model (*4*). When the response time is low, objective *C* suggests that Znn.com should decrement its server pool size (*2*) if it is near budget limit; objective *B* suggests that Znn.com should switch the servers to multimedia mode (*3*) if they are not already in that mode. When the response time is in the normal range, objective *B* suggests that Znn.com should switch the servers to multimedia mode if they are currently textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

Various adaptation strategies can be defined from these four tactics, and we have defined four, listed in Appendix C on page 184, and summarized here:

SimpleReduceResponseTime: When any client experiences a request-response time above threshold, this strategy lowers content fidelity by one level, then lowers fidelity again if the first attempt fails to bring response time below threshold.

SmarterReduceResponseTime: Let *n* be the count of clients experiencing request-response time above threshold; if *n* exceeds a tolerable percentage of total, this strategy enlists a server, then enlists another server, then lowers the fidelity one level, then repeats the last two sequence twice until successful.

ReduceOverallCost: When the total server cost exceeds a threshold value, this strategy discharges up to four servers, one at a time, until the cost is reduced below threshold.

ImproveOverallFidelity: When the average content fidelity of the servers drop below a threshold value, this strategy raises the fidelity level for all servers, up to twice, until the average fidelity rises above threshold.

Note that these have juxtaposing effects to allow system adaptation to balance overall objectives.

6.5.3 Experimental Setup

To demonstrate self-adaptation on a news website as motivated in Section 6.5.1, we configured a Znn.com system using open-source, commercial software, customized an instance of Rainbow on the system as described in the previous subsection, then performed a Slashdot-effect experiment with the system to determine the effectiveness of Rainbow self-adaptation.

The setup consisted of a pool of four typical Intel (~1 GHz) machines, each running a Debian-flavor Linux operating system, configured with an instance of Apache webserver. A fifth machine ran a load balancer to forward incoming requests in a round-robbin fashion to any active server among the four. Two additional machines were set up to act as the clients, using Apache JMeter, a Java application for testing web applications and measuring their performance, to simulate request loads from multiple clients.

To perform the experiment, we designed a workload that is characteristic of a *Slashdot effect* visitor traffic profile, as discussed in the next subsection, and devised the following trial types to test the hypotheses and assess Rainbow's effectiveness:

1. Control runs without Rainbow adaptation—to establish baseline and comparison envelopes
   (a) Nominal configuration: single active server with full-fidelity (multimedia) content
   (b) Maximum fidelity, highest cost: all servers active with full-fidelity content

(c) Maximum service capacity, minimum fidelity, highest cost: all servers active with lowest-fidelity (textual) content

2. Experimental runs with Rainbow adaptation

   (a) Rainbow self-adaptation capabilities tailored to Znn.com

      i. SimpleReduceResponseTime and SmarterReduceResponseTime strategies only—fidelity and cost dimensions to cope with *Slashdot effect*

      ii. Add ReduceOverallCost and ImproveOverallFidelity strategies—recovery from degraded modes; to illustrate full trade-offs

   (b) Monitoring only—to allow accounting for resource overhead of system monitoring

For each trial type, we performed five runs to smooth stochastics and to yield consistent outcomes. For every run, we collect statistics on the total number of samples, response latency measurements, request throughput, and any errors. We also track the corresponding cost and content fidelity values to compute cumulative utility and provide a complete picture of the trade-off space as defined by the overall objectives. We use the cumulative utility value as measure to assess variance and determine the final count of trial runs to perform.

6.5.4 *Slashdot Effect* Traffic Profile

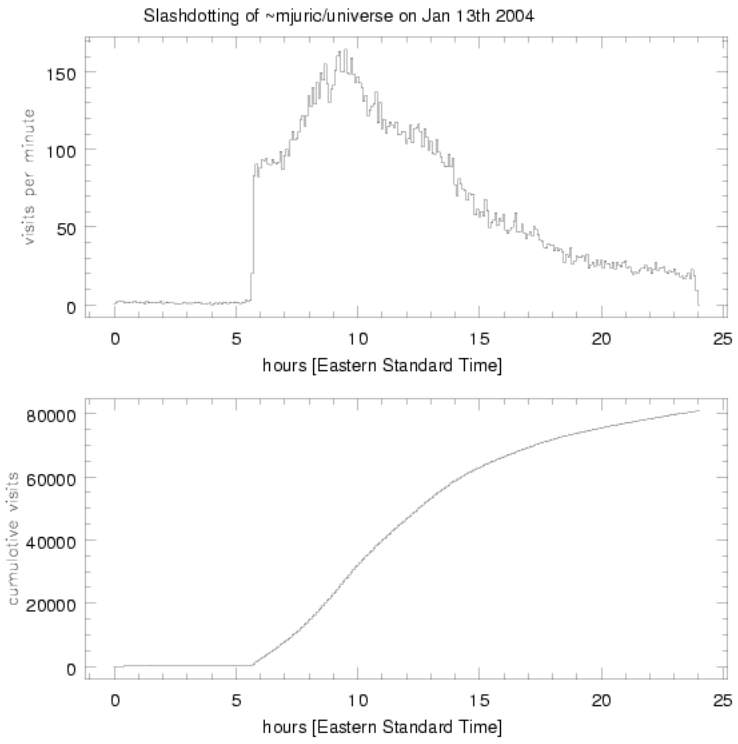


Figure 6.11: Graph of actual, peak-day traffic of a site experiencing *Slashdot effect*.

The sharp increase in visitor traffic from the *Slashdot effect* can range from several hours to a few days. The short Wikipedia article on this topic indicates that traffic might remain elevated for 12 to 18 hours until the posted Slashdot article is pushed off the front page. The article also graphically depicts a typical *Slashdot effect* traffic profile, marked by a sharp increase in traffic preceded by relatively low activity and followed by a gradually tapering tail [Wik08d]. Example of an actual *Slashdot effect* traffic is duplicated in Figure 6.11 [Jur04].

Due to the resource-demanding nature of the *Slashdot effect*, an adaptation that does not quickly offset the sudden rise in demand for resources would not be effective. Therefore, the initial sharp rise in traffic entails the critical duration of interest for our experiment purposes. For measurement purposes, we choose to observe a sustained duration after the initial rise to make sure that any effective adaptations remain effective for a reasonable amount of time. In lieu of 12 to 18 hours of actual traffic, we devise a *Slashdot effect* traffic profile patterned after the ~mjuric profile (Figure 6.11), scaled down to one hour (12:1) but kept at a similarly high visit rate:

1. 1 minutes of low activity, 6 unique visits/min
2. 5 minutes of sharp rise in requests, ramp up to 600 visits/min (+120 visits/min/min)
3. 18 minutes of peak in requests, sustained at 600 visits/min
4. 36 minutes of linear decrease, ramp down to 60 visits/min (-15 visits/min/min)

We constructed this workload in JMeter, using Gaussian random timer between requests. We then deployed this workload on two JMeter instances to generate the news reader traffic for our Znn.com example. Data collected from the experiment runs is summarized in the next subsection.

6.5.5 Data and Results

Figure 6.12 shows a graph for two experiment runs, control (red) versus adaptation using one simple strategy (green). For each run, the JMeter data table shows the total count of request samples, the statistics of request-response time, and the net throughput. The graph plots the latency per request, and the small table summarizes how many requests yielded latencies above 10 seconds and 1 second. The data indicated that Znn.com with Rainbow adaptation, in contrast to Znn.com without, yielded far lower latencies (902 of 1200, or 75%, requests served within 1 second vs. 80 of 1200, or 7%) and better throughput (3.5x of control).

While instantiating Rainbow for the Znn.com system, we tracked Rainbow customization activities in detail, shown below.[4] The customization effort, including architecture modeling, adaptation scripting, and development and testing of probes, gauges, and effectors, accounted for a total of **93.4 hours,** or approximately 2 1/3 work weeks. Of this, 13.3 hrs (14%) were used to describe the model, 49.1 hrs (53%) to develop probes and gauges, 7 hrs (8%) to develop effectors, 21.3 hrs (23%) to compose adaptation scripts, and 2.7 hrs (3%) to compile the customization files. Note that while the majority of the effort was spent developing monitoring capabilities, the resulting probes and gauges are reusable artifacts, so less such effort would be required as more are developed. Furthermore, the order of magnitude of effort has greater significance than the actual durations: note that most activities required on the order of minutes to a couple hours, not

[4][]: bracketed time indicates overlap with another time block, and excluded from total.

Aggregate JMeter Data									
		ms response							
URL	#Samples	Average	Median	90% Line	Min	Max	Error%	Throughput	KB/sec
0415K No Rainbow	1200	7981	1953	37514	93	52201	0.00%	8.4/sec	6.92
0415L SimpleReduce1	1200	1502	16	2187	0	22202	0.17%	29.5/sec	78.26

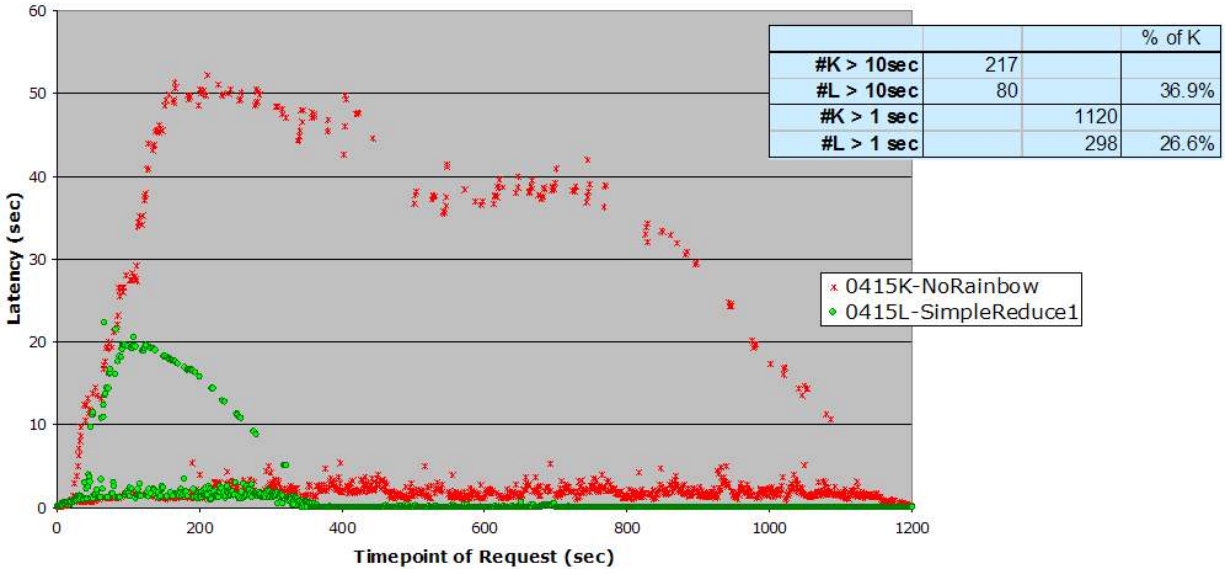


Figure 6.12: Graph summarizing preliminary Znn.com experiment data

days, while incremental changes required on the order of tens of minutes, not hours. We evaluate the cost-effectiveness of Rainbow adaptation engineering in Section 7.2.

#	Activity Description	Dur (hr)
	Rainbow Model Description	
K1	Created simple version of architecture model	1.0
K2	Developed full version of architecture model	8.0
K3	Added fidelity and cost thresholds	0.33
K4	Added request rate and other ApacheTop properties	0.67
K5	Added another constraint	0.17
K6	Added component deployment location property	1.0
K7	Added latency property to HTTP connector (follows M11)	0.17
K8	Created EnvModel type and incorporated with arch	2.0
K9	Added Client reverse constraint while testing strategies (incl. in S6)	[0.17]
	Subtotal Rainbow model description time:	**13.3**
	Probes and Gauges Development	
M1	ApacheTop utility Rainbow customization	20.0
M2	Fidelity detection perl script + Gauge (incl. in M3)	[3.0]
M3	CPU Load perl script (incl. learn /proc/stat) + Gauge	8.0

M4	ApacheTopGauge (mostly regex patterns)	4.0
M5	Rainbow probes perl module + sockets	4.0
M6	ApacheTop perl script (incl. in M5)	[2.0]
M7	DiskIOProbe perl script, via /proc/diskstats	2.25
M8	DiskIOProbe perl script, via iostat	0.75
M9	DiskIOGauge, aggregating stat from diskstat probe	1.5
M10	PingRTTProbe, java-based (mostly research time)	2.33
M11	LatencyGauge, using PingRTTProbe, est 1 KB data	0.25
M12	RtLatencyMultiHostGauge, gauges multiple hosts (added support framework)	4.0
M13	RtLatencyRateMultiHostGauge, 1st derivative	0.83
M14	ClientProxyProbe, to check experienced resp time	0.83
M15	End2EndRespTimeGauge, to check experienced resp time using ClientProxyProbe	[0.33]
	Subtotal probes and gauges development time:	**49.1**

	Effectors Development	
A1	Change server fidelity + Apache fidelity conf files	4.0
A2	Turn server on/off	2.0
A3	Set random reject	1.0
	Subtotal effectors development time:	**7.0**

	Stitch Scripts Composition	
S1	Created newssite Strategies and Tactics scripts	10
S2	Added newssite strategies and tactics	3.0
S3	Updated newssite strategies, added simpleReduce	0.5
S4	Modified newssite strategies, added variedReduce; Strategy testing by simulation (1)	1.5
S5	Strategy (*smarter*) testing by simulation (2)	1.0
S6	Strategy testing by simulation (3)	3.5
S7	Improved newssite raiseFidelity tactic, fix oscillation	0.75
S8	Modified newssite strategies for grammar update	1.0
	Subtotal Stitch scripts composition time:	**21.3**

	Rainbow Customization Files	(min)
R1	Gauge spec, add DiskIOGaugeT and instance	10
R2	Gauge spec, add LatencyGaugeT and instance	7
R3	Probe spec, add multi-host key-values	17
R4	Gauge spec, modified LatencyGaugeT to use Multi	11
R5	Gauge spec, add LatencyRateGaugeT and instance	10
R6	- Getting LatencyRateGaugeT to work	84
R7	Probe spec, add ClientProxyProbe (incl. in M15)	[5]
R8	Gauge spec, add ResponseTimeGaugeT and inst (incl. in M15)	[15]
R9	- Getting ReponseTimeGaugeT to work	20
	Subtotal Rainbow customization files time:	**2.7 h**

	Total Rainbow customization time:	**93.4 h**

6.5.6 Znn.com Summary

In summary, as evidence toward **V1–5**, the Znn.com example demonstrated the comprehensive instantiation of an integrated Rainbow framework with its customization points, including the use of an environment model. The overall customization effort required less than 94 hours, half of which may be amortized with reusable probes and gauges. Furthermore, the resulting self-adaptive Znn.com system demonstrated that, given a set of quality objectives, utility profiles, and preferences, Rainbow can select the best adaptation strategy that trades off across multiple quality dimensions to adapt the target system and achieve near-optimal overall utility. Finally, Znn.com exercised the capabilities of the Stitch self-adaptation language to represent four quality dimensions, two adaptation conditions, three different scenarios of utility preferences for testing (see Appendix C on page 188, under the heading "weights"), three operators, four tactics with cost-benefit attributes, and five strategies with conditions of applicability on style as well as system conditions, timing delay, and branch probabilities.

6.6 Interview with System Administrators

The design of the Stitch language was derived, in part, from our understanding of how sys-admins perform adaptive administrative tasks when they encounter system problems. We developed this insight over time from personal experiences and interactions with other sys-admins [Alm06, Bis06, Che08a, Whi06, You07, Rho08]. While our insight helped to shape the design of Stitch, to further substantiate the expressiveness Stitch and the suitability of its design, we arranged to interview sys-admins to gather systematic evidence [Alm06, Bis06, Whi06].

In this section we detail an interview process and a set of interview questions we developed. We then report results from two sys-admins, one from interview, another, self-guided decision analysis. Most of this content derived from the work of Ali Almossawi, an undergraduate student with prior experience as a sys-admin, who performed a summer independent study on the Rainbow project under my supervision. The objective of the interview process was to distill important self-adaptation concepts and determine mismatches of system-administrative process against Rainbow self-adaptation process.

6.6.1 Methodology

The interview process consisted of three steps. After initial contact with a sys-admin, we first prepared the sys-admin for the interview by asking a set of pre-interview questions that prime the sys-admin to think in terms of problem-objectives-actions, but without eliciting detailed responses. Second, we conducted the interview to elicit concrete responses and uncover in detail the mental process of system administration with respect to problem-solving courses of action. Third, we attempted to compose a script in Stitch for each of the problem-solutions discussed in the interview, in some cases interacting with the sys-admin to obtain feedback.

Pre-Interview To make the interview process constructive and efficient, the following pre-interview questions serve dual purposes: to set the interviewee with the proper frame-of-mind,

and to provide the interviewer with a high-level understanding of the system context within which the sys-admin works. Ideally, leave a hiatus of a few days to a week between receiving the pre-interview response and conducting the interview.

Context Briefly category the kinds of systems and applications you work with.

Problems Briefly describe some typical problems you encounter as a sys-admin.

Objectives For each of these problems, highlight any objectives you establish before attempting a solution, that is, indicate briefly what motivates your decisions.

Courses of action For each objective, briefly describe the abstract steps you take to achieve it.

Interview Allow approximately 40 minutes to conduct the interview. To guide the interview and maximize information elicitation, each of the following questions indicates an underlying purpose [P] before listing the question [Q]:

1. [P] In order to see how expressible is each adaptation step, we need to obtain, from *context*, an accurate description of the sys-admin's problem-solving process.
   [Q] Your pre-interview response identified a few problems and the associated problem-solving steps that you normally take. Could you walk me through each set of steps to show me your typical problem-solving workflow?
2. [P] In general, see whether utility theory is an appropriate decision technique; in particular, determine whether adaptation decisions are consistent (rational), an indication of whether quantifying factors using tactic attribute vectors is reasonable.
   [Q] When you encounter a problem, how do you determine what action to take and whether it is the best choice? In other words, how do you prioritize your choice? by simply picking the cheapest or sufficiently effective one, or using a more complex set of criteria?
3. [P] Determine if the decision-making process of the sys-admin is sufficiently rational to generalize as a yardstick for measuring the expressiveness of Stitch (need more than one interview). In fact, if some typical strategies of the sys-admin have been written down, they would serve as perfect specimens to gauge Stitch expressiveness or language omissions.
   [Q] Do you have any mitigation strategies written down that I can examine or do you usually rely on intuition? If intuition, would you say it's fairly consistent or does it change depending on case and time? Would you be able to codify your decision-making process or are there variable factors that require human judgment?
4. [P] Determine whether the sys-admin uses a catch-all solution (e.g., rebooting the server) or attempts solutions with partial confidence hoping that one of them fixes the problem; exploring this helps to classify the types of decision-making (e.g., pessimist or optimist).
   [Q] If you encounter a problem where you have insufficient information to determine how to fix it, what do you usually do?

6.6.2 Interview Results

Following this process, Almossawi conducted an interview on July 24, 2006, with Walter White, a former Carnegie Mellon University sys-admin. During a 20-minute interview, White described

three typical sets of problem-solutions. One set exhibited the highest relevance to illustrate system adaptation: administering students who abuse network bandwidth. Almossawi documented the interview, analyzed the sys-admin's solution strategy, and composed a Stitch script to codify the strategy and related tactics.

In a network bandwidth abuse case, a student had backed up ~80 GB of his hard disk onto his server space. The system administrator observed a spike in bandwidth usage the next day and a technician noticed that the backup tape was exhausted at around the same time. To prevent future repeat of similar situations, the sys-admin would ideally want the ability to track disk usage by file type, such as MP3s. The sys-admin also contemplated enforcing a disk quota on user accounts, but that policy would have unjustly prevented legitimate users from transferring large quantities of data; one solution would have been to enforce a disk quota only on the users with excessive usage. Finally, a monitoring capability with email notification might have alerted the sys-admin of the bandwidth abuse problem much earlier than the "next day."

Solution Strategy, Tactics, and Cost-Benefit Attributes Based on White's description of the usage abuse problem and potential solutions, Almossawi captured three adaptive concerns, a set of three tactics, and an overall strategy, summarized in Table 6.4. From his interview report, Almossawi noted that only one of the tactics was directly drawn from the interviewee, while the other two were inferred from context. Whether inferred or directly elicited, since these tactics fit the problem context, they serve our purpose to assess the expressiveness of Stitch.

Table 6.4: Summary of Walter's solution strategy and tactics

	[S] DealWithAbusiveUser *trigger: notable spike in bandwidth usage* *trigger: backup tape unexpectedly runs out*
	[T] increaseServerPoolSize (User user, Host h, int n) *guard: bandwidth is high* [overhead: 0.5; ill-feeling: 1; cost: 0]
	[T] banAbusiveUser (User user) *guard: bandwidth is high* *guard: the offense has been committed n times (where n>1)* *guard: the disk space usage is high* [overhead: 1; ill-feeling: 0; cost: 1]
	[T] warnAbusiveUser (User user) *guard: bandwidth is high* *guard: the offense has been committed once before* *guard: the disk space usage is at least medium** [overhead: 1; ill-feeling: 0.5; cost: 1]

The three adaptation objectives are shown below, where the *cost* attribute captures a potential for dollar value lost when executing a particular tactic.

1. Overhead: 1 if low, 0.5 if medium, 0 if high
2. Ill-feeling: 1 if low, 0.5 if medium, 0 if high
3. Cost (dollars): 1 if low, 0.5 if medium, 0 if high

To codify the adaptation strategy for this scenario, we establish reasonable assumptions[5] about the target system, which we can characterize as a set of servers (ServerT) and user machines (HostT) inter-connected by network links (LinkT). In addition, users (UserT) in the environment interact with the servers via machines they own. Using prior knowledge about the system, we can deploy probes to monitor system properties such as bandwidth usage by links and by hosts, host ownership, and disk usage by users. In a hypothetical model of the target system, we can track these properties with gauges and specify threshold values. In addition, we model the environment by tracking the users and their statistics, including offense history, which indicates a user's frequency of abusing the network. Although Walter indicated that disk space usage is a non-issue, we use it as our action guard to illustrate variability.

Stitch Script From his interview analysis, Almossawi composed a Stitch script, but we refined and improved upon it to better illustrate the expressiveness of Stitch. Except for minor adjustments,[6] the codified tactics correspond to what is described in the table above.

```
1  module abusiveUsers;
2  import model "TargetSys.acme" { TargetSys as M, TargetFam as T };
3  import model "TargetEnv.acme" { TargetEnv as E};
4  import op "example.operator.EffectOp" { EffectOp as S };
5  import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
6
7  define boolean styleApplies = Model.hasTypes(E, "UserT")
8      && Model.hasTypes(M, {"HostT","ServerT","SvrGrpT","LinkT"});
9  define set links = { select cn : T.LinkT in M.connectors | true };
10 define float bwUsage = Model.sumOverProperty("bandwidthUsed", links);
11 define set backupServers = { select c : T.ServerT in M.components | c.doesBackup };
12 define boolean hasViolation = bwUsage > M.HI_BANDWIDTH
13     && exists s : T.ServerT in backupServers | s.tapesAvailable = 0;
14
15 tactic increaseServerPoolSize (set hosts, int n) {
16   condition {
17     bwUsage > M.HI_BANDWIDTH;
18   }
19   action {
20     // acquire set of server groups to which the hosts connect
21     set groups = { select g : T.SvrGrpT in M.components
22       | exists h : T.HostT in hosts | Model.connected(h,g)};
23     for (T.SvrGrpT g : groups) {  // increase size of that group/pool
24       S.addServers(g, n)
25     }
26   }
27   effect {
28     bwUsage <= M.HI_BANDWIDTH;
29   }
30 }
31
32 tactic banAbusiveUser (set users) {
33   condition {
34     bwUsage > M.HI_BANDWIDTH;
```

[5] As demonstrated in prior examples, these assumptions are reasonable for Rainbow's capabilities.

[6] All tactics are parametrized with a set instead of a single element, and the disk space usage guard for the warnAbusiveUsers tactic is captured in the set-selection predicate.

```
35      exists u : E.UserT in users
36        | u.diskUsage > M.HI_DISK_USAGE && u.offenses > M.OFFENSE_THRESHOLD;
37    }
38    action {
39      set tgtUsers = { select u : E.UserT in users
40        | u.diskUsage > M.HI_DISK_USAGE && u.offenses > M.OFFENSE_THRESHOLD};
41      for (E.UserT u : tgtUsers) {
42        // lock user out of server space and blacklist his/her IP address
43        S.setPermissions(u, M.PERMISSION_BAN);
44        set hosts = { select h : T.HostT in M.components | h.ownerID == u.userID };
45        for (T.HostT h : hosts) {
46            S.banIP(h.ip);
47        }
48      }
49    }
50    effect {
51      bwUsage <= M.HI_BANDWIDTH;
52    }
53  }
54
55  tactic warnAbusiveUser (set users) {
56    condition {
57      bwUsage > M.HI_BANDWIDTH;
58      forall u : E.UserT in users | u.offenses <= M.OFFENSE_THRESHOLD;
59    }
60    action {
61      set tgtUsers = { select u : E.UserT in users | u.diskUsage > M.MED_DISK_USAGE};
62      for (E.UserT u : tgtUsers) {
63        // reprimand user by email and apply temporary quota to user account
64        S.setPermissions(u, M.PERMISSION_WARN);
65        S.warnUser(u);
66      }
67    }
68    effect {
69      bwUsage <= M.HI_BANDWIDTH;
70    }
71  }
72
73  strategy DealWithAbusiveUsers
74  [ styleApplies && hasViolation ] {
75    define abusiveHosts = { select h : T.HostT in M.components
76      | h.bandwidthUsed > M.HOST_BW_QUOTA };
77    define abusiveUsers = { select u : E.UserT in E.components
78      | exists h : T.HostT in abusiveHosts | h.ownerID == u.userID };
79    define shouldBanAny = exists u : E.UserT in abusiveUsers
80      | u.offenses > M.OFFENSE_THRESHOLD;
81
82    t1 : (bwUsage > M.HI_BANDWIDTH) -> increaseServerPoolSize(abusiveHosts, 1) {
83      t1a : (bwUsage <= M.HI_BANDWIDTH) -> done;
84    }
85    t2 : (shouldBanAny) -> banAbusiveUser(abusiveUsers) {
86      t2b : (bwUsage <= M.HI_BANDWIDTH) -> done;
87    }
88    t3 : (default) -> warnAbusiveUser(abusiveUsers) {
89      t3b : (bwUsage <= M.HI_BANDWIDTH) -> done;
90    }
91  }
```

The codified strategy, DealWithAbusiveUsers, specifies two conditions of applicability: (a) a number of types have to be defined in the style, and (b) the system must be in a state of violation. Once these conditions are satisfied and the strategy is chosen, one of three paths is taken depending on whether the link bandwidth usage exceeds a high threshold, whether any user should be banned, or otherwise. As a final point of illustration, the script uses both the architecture as

well as environment models, and is able to express predicates that reference elements simultaneously from both models. This expressiveness empowers the adaptation engineer to reason about adaptations by combining information from both the architecture and the environment.

6.6.3 Adaptation Analysis from Almossawi's Administrative Experiences

As additional supporting evidence, Almossawi documented two scenarios from his personal experiences as a sys-admin. Because he already understood the constructs of Stitch, Almossawi structured the scenarios to facilitate representation in Stitch, though no actual Stitch scripts were specified. Almossawi also documented two primary decision criteria for choosing appropriate courses of action to fix a system problem:

> *How quickly can I execute it*? I try to pick the course of action that requires the least amount of effort. *How severe is the situation*? If someone discovers a critical exploit in a software that I have installed and tells the whole world about it via, say, BugTraq, I'll probably immediately take the webserver down, apply the patch, then bring it back up again. If one isn't available, I'll most likely take just the application down until one is released.

To combat a noticeable slowdown in accessing web pages on his server:

```
|_ Ping and traceroute the webserver
  |_ If reply time is too long or the request times out
    |_ Check web host's network usage graphs
    |_ Contact network administrator
|_ Check server load
  |_ If high
    |_ Quick fix
      |_ Restart the httpd process
      |_ Restart the mysqld process
    |_ Check processes
      |_ Check for offensive running processes
        |_ Kill any that I find
    |_ Check error logs
        |_ Increase value of ServerLimit and MaxClients in httpd.conf
        |_ Restart the httpd process
    |_ Check disk space
        |_ If < 1% free
          |_ Delete rotated log files
  |_ If nothing works, check U.S. time
    |_ If before 11 or after 6 or if slowdown is severe (website's peak time)
      |_ Reboot webserver immediately
    |_ Otherwise, wait until that timeframe then reboot
    |_ If problem still persists
      |_ Contact web host's tech support
```

To combat possible suspicious activity on his server:

```
|_ Logwatch email indicates suspicious IP address or brute force attempt
  |_ Add IP address to firewall's "deny" list
  |_ Flush firewall's rules then restart it
  |_ Check server logs and look up IP address to see if it has a history
    |_ If it does
      |_ Change superuser's password
  |_ Check server logs to ensure IP address didn't gain access to the system
    |_ If I suspect it did
      |_ Immediately run rootkit checker
      |_ Immediately run anti-virus scanner
```

```
|_ If for some reason I'm still suspicious
  |_ Backup all data
  |_ Reformat webserver
  |_ Restore data
```

6.6.4 Interview Summary

Based on Almossawi's interview and analysis results, the concerns, problems, and solutions expressed by both the sys-admin interviewed and by Almossawi himself appeared to corroborate concepts embodied in Stitch. In particular, evident in the documented responses were elements of objectives, observable system conditions, specific actions in response to specific conditions, and, to a lesser extent, preferences. Assessed abstractly, Stitch provides the appropriate constructs and has the expressiveness for capturing an adaptive administrative strategy concisely and intuitively. We evaluate the expressiveness of Stitch in greater detail in Chapter 7.

6.7 Real-World Adaptive Scripts in Stitch

Carnegie Mellon University possesses a sophisticated campus computer network. Over the past decade, CMU has invested extensive engineering efforts to improve its networking infrastructure and automate system administration [You07], making it a prime candidate to find evidence of system adaptation scripts. Of the network administrative subsystems, I investigated the network bandwidth enforcement (*netbwe*) subsystem, acquiring its Perl source code in January 2008 from the network administrators [Rho08].

The *netbwe* subsystem is executed daily (as a cron job) to monitor network bandwidth usage and enforce quota by machine, using sensors installed on campus routers. With a database, *netbwe* tracks usage history, records quota violation, and tracks violation states. It interacts with the notification subsystem (*epidemic*) to alert offending machine owners by email, and interacts with a *netblock* component to block network access for repeat offenders. In short, *netbwe* has the monitoring, detection, decision, and action elements of a self-adaptive system. Figure 6.13 illustrates the system context of the *netbwe* subsystem.

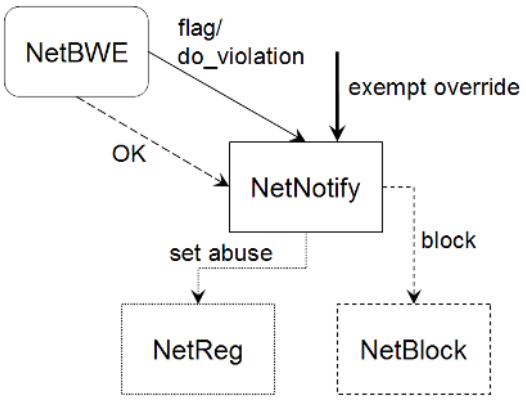


Figure 6.13: A system context diagram of the *netbwe* subsystem

Given that *netbwe*, a bona fide system administrative subsystem, fits the profile of a self-adaptive system, the ability to express this adaptive script in Stitch would strengthen the case of its expressiveness. Prior to examining the *netbwe* source code, I formulated the following set of hypotheses, which also provided a concrete to-do checklist for this endeavor:

- Logical adaptive task units are *distinguishable*: operator, tactic, strategy.
- We can identify *core commands* as operators.
- We can identify frequently used *subroutines* as tactics, the adaptation steps, with conditions of applicability and intended effects.
- We can identify *top-level adaptation decisions*, consisting of observable conditions and decision-points, as strategies.
- An *architectural model* of the target system provides a core structure to reason about adaptation concerns, including what components to monitor, what properties can be monitored, and what changes can be effected.

The *netbwe* subsystem consists of 19 Perl source files with about 10k physical source lines of code (SLOC).[7] In particular, three frontend Perl programs and two backend Perl module subroutines comprise the core adaptation functionality, while the remaining subroutines would be considered effectors with *operator* counterparts for the adaptation script. The five programs/subroutines are summarized below and detailed in Appendix D.2 on page 192:

load_state loads network usage states by machine from bandwidth logs

log_usage determines and records current usage against database of prior machine states

do_violation determines machine violation incident, notifies owner, records violation state

Netbwe::API::violations collects list of hosts whose states reflect bandwidth usage violations

Netbwe::API::log_violation determines if any violation has occurred and LOG it to *netnotify*

After analyzing the *netbwe* subsystem, parallel customizations for Rainbow are identified.

Adaptation Objectives and Conditions From conversations with the network admins, the *netbwe* subsystem has these adaptation objectives:

- Fair usage
- System fit-for-purpose (i.e., campus researchers can do what is needed)
- Reasonable campus connectivity provisioning cost, as determined by capacity and usage

These objectives correspond to the following adaptation conditions:

- Threshold on bandwidth usage
- Exemptions and dated restoration of violation states

Model Modeling the target system requires modeling the architecture and environment of CMU's network infrastructure, as illustrated in Figure 6.14.

[7]The report is generated using David A. Wheeler's 'SLOCCount'.

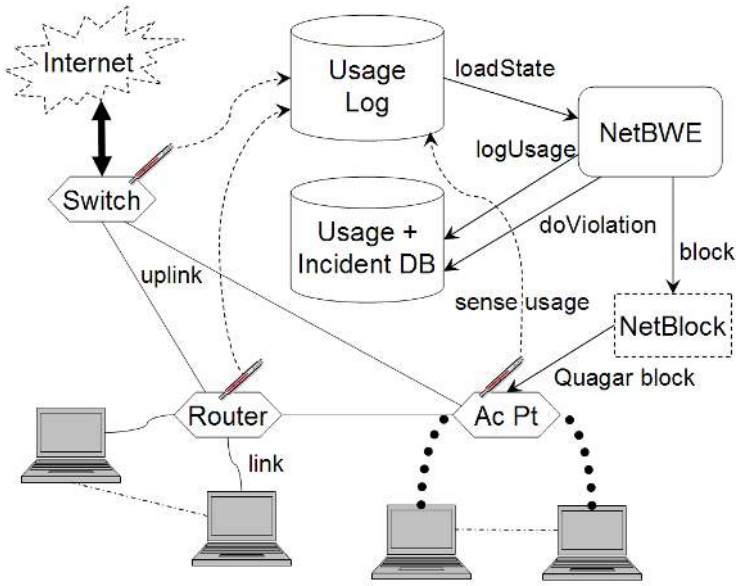


Figure 6.14: Model of the CMU network infrastructure

The architectural style would define component types SwitchT, RouterT, AccessPointT, and HostMachineT, and connector types LanLinkT and WirelessLinkT. The style would also define operators including machineModify(HostMachineT,...) and netBlock(RouterT,...). Most important, HostMachineT would define specific properties to track machine states:

- Machine/netreg ID
- Quota category ID
- IP/MAC address
- Daily in/out rate
- Daily inbound/outbound usage
- Total rate
- State of violation
- Exemption status
- Primary user

The architecture model would consist of link-connected instances of the defined component types, most notably HostMachineT. The environment model would consist of (a) users with user properties, (b) admins, and (c) model references between the users and the machines they own. The admins and users serve as targets for *netnotify*. Note that routers can be modeled as part of the system or as resources in the environment, and we have opted to do the former because it interacts directly with other system elements and it has a defined operator of state modification.

Probes and Gauges The load_state program serves the probe function to get system states from sensors. The log_usage program serves the gauge function to interpret and abstract states to populate the architecture model (the database).

Operators, Tactics, Strategies Some of the *netbwe* Perl subroutines act as the effector counterparts of adaptation operators, while others serve as library utilities. We import the latter to reuse the Perl modules; examples include db-connect, db-add, parse-message, send-mail, event_list, and machine_list. For Rainbow customization, the required operators would be translated to effectors realized by the corresponding Perl subroutines. By analyzing the adaptation objectives in combination with the Perl subroutines, we identified four strategies and five required tactics, with a few of them codified in Stitch:

```
1  // Common script content for strategy and tactic descriptions below.
2  module netbwe.tactics;
3  import model "CmuNetworkSys.acme" { CmuNetworkSys as M, CmuNetworkSys as T };
4  import op "netbwe.operator.ArchOp" { ArchOp as S };
5  import op "perl.module.NetbweUtil";        // Netbwe Perl module
6  import op "org.sa.rainbow.stitch.lib.*";   // Model, Set, & Util
7
8  define boolean styleApplies = Model.hasType(M, "HostMachineT");
9  define boolean hasViolations = exists m : T.HostMachineT in M.components
10     | NetbweUtil.usageRate(m, Util.today()) > M.quotaMax(m.quotaID);
11 // assuming an "admin" component instance exists in the arch model...
12 define boolean hasExemptionOverrides = Set.size(M.admin.newExemptSet) > 0;
```

Strategy EscalateViolationStates escalates violation states of machines exceeding daily quota.

```
1  strategy EscalateViolationStates
2  [ styleApplies && hasViolations ] {
3    // captures logic in lines 130-198 of "API::violations" in App D.2, excerpted below
4    define set{T.HostMachineT} machsInViolation = { select m : T.HostMachineT
5        in M.components
6        | NetbweUtil.usageRate(m, Util.today()) > M.quotaMax(m.quotaID) && !m.exempt };
7    define cond = Set.size(machsInViolation) > 0;
8
9    t0: (cond) -> markViolation(machsInViolation) {
10     t1: (cond) -> emailNotify(machsInViolation) | done;
11     t2: (default) -> TNULL;
12   }
13 }
```

```
// lines 130-198, note the checking of quota limit and exempt status
  # Process the list
  foreach my $key ( keys %$mach ) {
    # flush out the ones that are not over quota
    if ( $mach->{$key}{Rate} <= $limit ) {
      delete $mach->{$key};
      next;
    }
    # Drop any machine that is exempt (****)
    my $query; ...
    my($status, $reason, $extra) = CMU::Netbwe::ChkStatus($dbh, $user, $query, $qlookup);
    if ( $status eq "exempted" ) {
      delete $mach->{$key};
      next;
    ...
    }
    ...
    # if commit is yes, then log the violation to netnotify
    ( $vio, $reason ) = log_violation( ... )
      if ( $commit eq 'yes' );
    ...
    if ( ( defined $vio ) && ( ref $vio ) ) {
      $mach->{$key}{result} = $vio;
    }
  }
```

Strategy RestoreViolationStates reverses violation states of machines that pass probation.

Strategy BlockMachines blocks network access of machines with a "ban" violation state.

Strategy ExemptMachines adjusts violation states of machines with exemption override.

```
1  strategy ExemptMachines
2  [ styleApplies && hasExemptionOverrides ] {
3    define set{T.HostMachineT} machsToExempt = { select m : T.HostMachineT in M.components
4        | Set.memberOf(m.id, M.admin.newExemptSet) };
5    define cond = Set.size(machsToExempt) > 0;
6
7    t0: (cond) -> setExempt(machsToExempt) {
8      t1: (cond) -> modifyMachines(machsToExempt, "exempted") | done;
9      t2: (default) -> TNULL;
10   }
11 }
```

Tactic markViolation marks machines with the next state of violation.

```
1  // extracted from Perl program "log_violation"
2  tactic markViolation (set{T.HostMachineT} machs) {
3    condition {
4      forall m : T.HostMachineT in machs | NetbweUtil.hasViolationRecord(m.id, m.quotaID);
5    }
6    action {
7      for (T.HostMachineT m : machs) {
8        // create a violation log entry (lines 26-40,80-84 in App D.2, excerpted below)
9        record vioEnt = [ /*... transfer machine props, e.g., m.id ...*/ ];
10       // determine next violation state (lines 72-77)
11       int nextState = NetbweUtil.computeNextState(m.violationStatus);
12       vioEnt.violationStatus = nextState;
13       // modify the machine with new states (line 90)
14       S.machineModify(m, vioEnt);
15     }
16   }
17   effect {
18     true;
19   }
20 }
```

```
// lines 26-40
    # log a new entry for them...
    $vals{'machines.mac_address'} = $svals->{'machines.mac_address'}
      if ( defined $svals->{'machines.mac_address'} );
    $vals{'machines.mac_address'} = $svals->{mac_address}
      if ( defined $svals->{mac_address} );
    $vals{'machines.ip_address'} = $svals->{ip_address}
      if ( defined $svals->{ip_address} );
    ...
    $vals{'machines.quotaID'} = $quota;
    $vals{'OUTBOUND_USAGE'}   = sprintf( "%-8.3f", $outbound / 1024 );
    $vals{'INBOUND_USAGE'}    = sprintf( "%-8.3f", $inbound / 1024 );
// lines 80-84
      $vals{'machines.status'} = $ttimes->{ $data->[1][ $dapos->{'machines.status'} ] }{
          promoteto};
      $vals{'machines.status_date'} = $date;
      $vals{'machines.reason'} = "Exceeded $mode quota (transmitted $outbound Mbytes
          outbound, $inbound Mbytes inbound) state $vals{'machines.status'} on $date";
      $vals{'OUTBOUND_USAGE'} = sprintf( "%-8.3f", $outbound / 1024 );
      $vals{'INBOUND_USAGE'}  = sprintf( "%-8.3f", $inbound / 1024 );
// lines 72-77
    #  Get the list of state change times for this quota
    ( $ttimes, $reason ) = full_state_list_hash( $dbh, $user, $quota );
```

```
    ...
    $next_state = $ttimes->{$data->[1][$dapos->{'machines.status'}]}{promoteto};
// line 90
        ( $data, $reason ) = machine_modify( $dbh, $user, $data->[1][ $dapos->{'machines.
            id'} ], $data->[1][ $dapos->{'machines.version'} ], \%vals, $mail );
```

Tactic emailNotify notifies machine owners of quota violation.

```
1  // extracted from Perl program "do_violation"
2  tactic emailNotify (set{T.HostMachineT} machs) {
3    string abuseDlist = "post+org.acs.ng.abuse@andrew.cmu.edu";
4    string bwDlist = "post+org.acs.ng.project.bandwidth@andrew.cmu.edu";
5    condition {
6      forall m : T.HostMachineT in machs | NetbweUtil.hasViolationRecord(m.id, m.quotaID);
7    }
8    action {
9      string msg = /*...body text...*/;
10     NetbweUtil.sort(machs, "machines.hostname");   // sort by hostname
11     for (T.HostMachineT m : machs) {   // compose message per host (lines 58-63)
12       msg += NetbweUtil.swrite(m.hostname, NetbweUtil.long2dot(m.ipAddress),
13                                m.macAddress, m.violationStatus);
14     }
15     // send email off (line 65)
16     S.sendmail(M.admin, msg, abuseDlist, null, bwDlist);
17   }
18   effect {
19     true;
20   }
21 }
```

```
// lines 58-63
  foreach (@hosts) {
    $msg .= swrite(
  <<'END', $_->[ $hopos->{'machines.hostname'} ], CMU::Netbwe::long2dot( $_->[ $hopos->{'
      machines.ip_address'} ] ), $_->[ $hopos->{'machines.mac_address'} ], $_->[ $hopos
      ->{'machines.status'} ] );
  @<<<<<<<<<<<<<<<<<<< | @<<<<<<<<<<<<<< | @<<<<<<<<<<< | @<<<<<<<<<<<<<<<<<<<
END
  }
// line 65
  CMU::Netbwe::send_mail( $msg, 'post+org.acs.ng.abuse@andrew.cmu.edu', undef, 'post+org.
      acs.ng.project.bandwidth@andrew.cmu.edu' );
```

Tactic modifyMachines modifies the violation or exemption state of machines.

Tactic netblockMachines issues a block request on machines via *netblock*.

Tactic setExempt sets exempt privileges machines.

This exercise applied the Rainbow approach to the CMU network bandwidth enforcement scenario. We have shown that, by identifying and extracting parallel Rainbow adaptation elements from the scenario, Stitch is capable of representing the adaptation-oriented system administrative concerns as embodied in actual administrative Perl source scripts.

Furthermore, this exercise demonstrated additional benefits in representing the *netbwe* subsystem in Rainbow with Stitch. Representation in Rainbow clearly separates the concerns of adaptation, so that probes, gauges, effectors, and adaptation choices are not distributed throughout the code. In particular, adaptation choices are made prominent in strategies, rather than being buried deep inside an API subroutine (Netbwe::API::violations). Finally, the distinction between strategy and tactic enables the adaptation engineer to reason about and describe the specifics of an adaptation action as an intellectually separate process from deciding when to take each action.

6.8 Summary

In this chapter, we showed five instantiations of Rainbow. In terms of the validation points, these five example systems demonstrated the following.

V1 Single property in three styles: *performance* in CSSys, Znn.com, and Libra; *security* in UniversityGradeSys; *availability* in TalkShoe

V2 Three properties in one style: Znn.com with *performance*, *cost*, and *quality*

V3 U se of environment model: TalkShoe (admin entity); Znn.com (spare servers)

V4 I nfrastructure reuse and ease of customization: CSSys $\Rightarrow$ Libra; TalkShoe $\Rightarrow$ Znn.com

V5 M ulti-objective trade-off: Znn.com

In addition, interview with system administrators together with analysis of real administrative scripts showed the potential expressiveness of Stitch. In the next chapter, we evaluate the extent to which the Rainbow approach, supported by evidence from these five cases and the two system administrative example, fulfills the thesis claims.

Chapter 7

Thesis Evaluation

In Chapter 3 we enumerated the requirements of **generality**, **cost-effectiveness**, and **transparency** for a self-adaptation approach and described the overall Rainbow approach and high-level framework capabilities. In Chapter 4 we presented the features and semantics of the Stitch self-adaptation language, while in Chapter 5 we detailed how to customize the framework. Assessed abstractly, these chapters have addressed the requirements as follows:

- Rainbow's framework of mechanisms realizes the canonical self-adaptation control process exemplified by the IBM Autonomic Computing reference framework.
- For **generality**, the Rainbow approach leverages the notion of software architectural style to characterize and define explicit customization points for tailoring common, reusable infrastructures of the framework to specific styles for multiple quality concerns.
- For **cost-effectiveness**, the Rainbow approach prescribes an adaptation engineering process that guides a systematic, incremental customization of Rainbow to target systems.
- For **transparency**, the Rainbow approach provides a self-adaptation language that
    - separates the concerns of self-adaptation into distinct concepts to facilitate reasoning;
    - provides explicit constructs for those concepts to allow (a) representation of adaptation knowledge and (b) separating definitions according to domain expertise; and
    - enables quantifying the utility of adaptation with respect to business objectives to support (a) trade-off analysis of adaptation strategies and (b) integration of adaptation strategies from different domains to satisfy multiple objectives.

In Chapter 6, we enumerated five validation points to assess our approach and presented five instantiations of Rainbow as supporting evidence for validation. We further interviewed system administrators and translated real-world system adaptation scripts as support for the design and expressiveness of self-adaptation concepts in Stitch. Finally, we related each piece of evidence to the validation item that it supports. In this chapter, we summarize the pieces of supporting evidence and evaluate how each helps to satisfy the thesis claims introduced in Section 1.4.

7.1 Claim: Generality

As noted in Section 1.4.1, we evaluate the **generality** claim by

- Showing an in-depth demonstration of the Rainbow adaptation cycle on one system; and
- Describing how customizable pieces of the Rainbow framework are specified for representative styles of system and qualities of concern:
  - Styles: client-server, service-coalition, N-tier
  - Qualities: *performance*, *cost*, *content fidelity*, *availability*, *security*

The three styles—client-server, service-coalition, and N-tier—cover a majority of systems we care about. Client-server and N-tier systems, typified in the Garlan and Shaw software architecture book [SG96], pervade commercial information technology systems (e.g., J2EE family) and many user applications (e.g., instant-messaging). In particular, the N-tier style, with variations, represents the major style of architecture for backend computing infrastructures in e-commerce (e.g., Amazon, Ebay) and business information technology [BMR+96]. The service-coalition style encompasses peer-to-peer systems or, more specifically, systems composed from loose computing nodes providing functionalities as services.[1]

As suggested by the SEI report on quality attributes [BLKW97], in designing systems, modern system architects typically care about performance, security, and dependability (which encompasses availability). For performance, system engineers and users typically are concerned with service-time, latencies, and throughput of requests in the system. Additionally, system owners would be concerned with the cost to operate and maintain the system. Furthermore, particular to a system that provides content services, content fidelity and accessibility of the service constitute two more important concerns.

Presented in Section 6.5, the Znn.com instantiation of Rainbow demonstrates the full self-adaptation capabilities of the framework and details the customization of Rainbow, specifically:

- Typical e-commerce architectural style: N-tier system;
- Multiple objectives: performance, cost, content fidelity, and service disruption;
- All phases of the adaptation process: monitoring, detection, decision, and action;
- Effective self-adaptation of the target system that achieves the highest overall utility to satisfy multiple business objectives ; and
- Acceptably low resource overhead on CPU utilization and network bandwidth usage

The Client-server system described in Section 6.1 demonstrates Rainbow's applicability to a client-server style of system with *performance* as a quality concern. The Libra videoconferencing system described in Section 6.2 demonstrates Rainbow's applicability to a service-coalition-style system with *performance* and *cost* quality concerns. The university-grade system presented in Section 6.3 suggests Rainbow's applicability to a composite client-server-shared-data style of system with *security* as a quality concern. The TalkShoe infrastructure presented in Section 6.4

[1] Service-oriented architectures (SOA) would be considered a specialization of this style, with stronger composition constraints than *loose coalition*.

demonstrates Rainbow's applicability to an N-tier-style system with an *availability* quality concern. In addition, if we consider a network of servers and host machines hosting a set of application services (e.g., sshd, httpd, subversion) as a coalition of services, the scenarios described in Sections 6.6 on sys-admin experiences and 6.7 on *netbwe* also suggest Rainbow's applicability to a service-coalition style of systems with a dependability or availability quality concern.

Security is a complex quality attribute, pervading nearly all aspects of software design. Experts agree that support for security is best considered upfront in the design of a system, and not tacked on as an after-thought [Ram02]. Furthermore, most properties of security are difficult to quantify, even by security experts themselves [But02], so self-adaptation for security may appear irrelevant and infeasible.

However, experiences of security engineers suggest that being able to *adapt* a system to resolve security concerns could be a crucial or beneficial capability [GC03]. For instance, for performance reasons, a typical system cannot afford to have all security features enabled at all times. Ideally, more resource-intensive security capabilities, such as fine-grained network intrusion detection, would be enabled only as needed. Furthermore, as demonstrated by Butler using techniques developed from decision theory, some security properties can be quantified in relative terms, providing a means of making adaptation decisions [But02].

The example scenario in Section 6.3 explores the use of an essential security concern—relative security risk—to enable self-adaptation for security. By defining an architectural style that quantifies relative security risk, defining intrusion properties to monitor (e.g., intrusionProbability), identifying a candidate set of change operators and adaptation tactics, and composing example security-targeted adaptation strategies, we have shown the design of a customization of Rainbow that can potentially self-adapt a system for security to counter detectable intrusions.

Together with the Znn.com system in Section 6.5, these examples demonstrate Rainbow's applicability to three architectural styles and five quality attributes, providing evidence of coverage for a representative set of architectural styles and for typical qualities of concern.

7.2 Claim: Cost-Effectiveness

As noted in Section 1.4.1, we evaluate the **cost-effectiveness** claim for Rainbow by

- Showing that the framework is flexbily customizable to make a system self-adaptive; and
- Qualitatively assessing task-based effort-savings with the framework versus without.

To determine the cost-effectiveness of the Rainbow approach, it is important to assess the engineering effort for the generic framework separately from effort to engineer adaptation for a target system. Framework engineering provides the core self-adaptation functionalities, makes them generally available, and improves the common infrastructure shared by all customizations. The framework engineering effort manifests itself as an upfront cost with diminishing incremental cost and growing return-on-investment as customization instances increase [CN01]. The effort to engineer target-system adaptations, such as that for TalkShoe, focuses primarily on the domain-specific adaptation concerns and knowledge relevant to the target system. We therefore focus on two framework aspects of cost-effectiveness: *reuse* and *ease of use*.

For *reuse*, we assess the extent to which the common infrastructures and, as a second-order effect, the customized parts of the Rainbow framework can be reused from one instantiation to the next. A basic measure of engineering effort, the "source line of code" (SLoC), gives an indication of how much self-adaptation functionality an engineer would *no longer* need to implement, and would thus save, each time the Rainbow framework is used to add self-adaptation capabilities to a new system. The SLoC measurement is generated using David A. Wheeler's "SLOCCount" utility on the source code of the Rainbow framework runtime: as of December 2007, SLOCCount measured **24,891**[2] total physical source lines of Java code, debugged lines of code that excludes comment, blank, and non-essential lines. Excluding the initial prototyping and research time, these ~25 kSLoC were developed over a period of two years. We have excluded from this count various components of reusable code that we built on but did not develop as part of this thesis, including AcmeLib, the Stitch editor, and third-party event transport libraries.

For second-order reuse, as described in Chapter 5, the customization contents for each Rainbow instantiation consisted of Yaml and Acme specification files and target system-specific translator implementations (i.e., probes, gauges, and effectors). These customizations are further reusable in new instantiation efforts, and examples of such reuse appear later in this section. As more instantiations are done, customization contents can be accumulated in a library, further facilitating reuse, amortizing cost, and reducing effort.

For *ease of use*, we assess the degree of effort required to customize Rainbow and add self-adaptation to a target system. To evaluate effort savings, we decompose the self-adaptation engineering process into coarse-grained tasks and estimate the time to complete each task. Rainbow customization entails three development tasks and one evolution task carried out by adaptation engineers (cf., roles in Table 5.2 on page 71): *domain analysis*, *model capture*, *design and implementation*, and *updates and modifications*.

Domain analysis (D) In the domain-analysis stage, the adaptation integrator identifies business objectives to capture quality dimensions and utility preferences, and determines system monitoring (probes) and action (effectors) hooks for adaptation.

Model capture (M) In the model-capture stage, the adaptation integrator captures the architecture model for the target system, either reusing a style definition from a style library or working with the style writer to define a new style. The integrator may need to adapt and refine the style to fit the target system, in particular, to capture specific adaptation conditions. The integrator also determines the model properties (gauges) to monitor.

Design-implementation (I) In the design-implementation stage, the system adapter finds or develops probes and effectors; the tactic writer specifies tactics; the strategy writer composes strategies; the adaptation integrator specifies cost-benefit attributes for tactics, integrates the Rainbow parts, tests the adaptation roundtrip, and refines.

Update-modification (U*) In subsequent iterations of the update-modification stage, the adaptation engineers may add or evolve quality dimensions, utility preferences, properties to observe or states to change, and tactics and strategies. They also modify any customization elements affected as a result.

[2]While a custom implementation is likely to have less SLoC than a generic framework and yield less saving per instantiation, the overall saving would likely exceed the framework cost on the second or third instantiation.

Reuse The example systems provided two data points of framework reuse. The Rainbow prototype for the Client-Server system (Section 6.1) was almost entirely reused to add self-adaptation capabilities to the Libra videoconferencing system (see Section 6.2). As reported in [GCH+04], the Rainbow prototype consisted of the adaptation mechanism, model manager, gauge and probe infrastructures, gauges and probes, and translation and system-layer infrastructures, totaling 102 kilolines of code (KLoCs), of which 100 KLoCs (~98%) were reused. (This reuse figure includes both the core Rainbow infrastructure and the customization pieces, although the original numbers did not make this distinction.)

Using the engineered Rainbow framework, TalkShoe (Section 6.4) and Znn.com (Section 6.5) demonstrated a first-order reuse of the common infrastructures in entirety, approximately 25 KLoCs. In addition, second-order reuse of the customization elements includes the performance-related system probes and gauges, the architectural style description, and customization files. In principle, strategies, tactics, and the utility specification are also reusable. The Znn.com instantiation shows one example of such reuse: the strategy SimpleReduceResponseTime was reused from the first Client-Server system, with one minor modification to change content fidelity rather than switch server group. More generally, for target system with similar quality concerns, tactics can typically be reused with minor adjustments to the cost-benefit attributes, and strategies can be reused with minor modifications of matched conditions or invoked tactics.

Ease of Use In the TalkShoe instantiation of Rainbow reported in Section 6.4, the development time to customize Rainbow for the MP3 scenario totalled approximately 34 hours, 26 of which was attributable to architecture modeling and customization of Rainbow to the TalkShoe environment, including the development and testing of probes, gauges, and effectors. Because 14 of those hours, as well as the initial 8 hours, involved two persons, the customization effort required 40 man-hours while the overall scenario required 56 man-hours of effort. In the Znn.com instantiation of Rainbow reported in Section 6.5, the development time to customize Rainbow totalled approximately 93 hours, more time than TalkShoe, but also yielding significantly more self-adaptation components. Note that in both cases, customization efforts match closely with the best-case effort estimation in Table 7.1. (We discuss this observation further in Section 8.7.)

The Znn.com customization details—model, probes, gauges, effectors, Stitch scripts, utility and other specification files—illustrated clearly-defined customization points, repeatable process, and reusable artifacts. In addition, development data showed that adding support for a new quality concern incurred a low incremental cost, within hours to a few days, which entailed efforts to find or develop new probes and gauges, to enhance the architecture model, to modify existing customization files, and to add new or modify existing tactics and strategies. Thus, this evidence supports the last experimental hypothesis outlined in Section 6.5.1.

The development time data for TalkShoe reported in Table 6.2 on page 104, accounting for just the adaptation engineering efforts, indicated reasonably low effort required to customize Rainbow. Although a 40 man-hour effort, or 5 working days, might seem somewhat high for a simple MP3 recording scenario, there are three reasons why this development-time data is positive evidence for the cost-effectiveness of the Rainbow approach. First, because the MP3 scenario marked the first customization effort of Rainbow to TalkShoe, a large fraction of the time spent was on learning and knowledge transfer: the Rainbow approach for Bob, and domain

knowledge of the TalkShoe business for me. After the initial learning curve, most of the two-person effort was accomplished by a single person.

Second, as new adaptation scenarios are added for a particular target system, the incremental effort required to customize Rainbow decreases as more reusable pieces are created—probes, gauges, effectors, model elements, tactics and strategies. Third, without the Rainbow framework, the effort required to build similar monitoring, modeling, and effecting capabilities directly into the TalkShoe infrastructure would have been significantly higher. In addition, TalkShoe would have increased their upfront risks by exposing their infrastructure to adaptation logic that are custom-built and wired into their system code.

To compare efforts, let us explore a hypothetical scenario: assume that even without Rainbow the same probes, gauges, and effectors would have to be implemented to build the adaptation mechanism into TalkShoe, with a simpler reactive mechanism created in place of the architecture model and the adaptation manager. Assume also that only half as much time would be spent on simpler reactive mechanism as on developing the architecture model (6h) and the Stitch scripts (8h), that is, 7 hours. Additionally, we can reasonably expect that similar deployment and roundtrip debug efforts would be required. Thus, without the initial 3 hours of environment setup and 7 hours from the reactive mechanisms, the hypothetical scenario would most likely have required a single engineer ~24 hours to complete.

However, this estimate does not account for the effort necessary for the basic adaptation plumbing. Based on the Rainbow framework development experience, it would take over 2.5 months to design and implement the communication infrastructure and plumbing for the probes, gauges, and effectors. Even if we assume simpler requirements for the hypothetical case, incurring only *half* the time to develop Rainbow, it would still yield a total effort of more than one month to add adaptation capabilities into TalkShoe. If we translate this effort to ~40 working days, that would mean nearly an order of magnitude (~8x) increase in development effort to directly build the adaptation capabilities into the TalkShoe infrastructure. Furthermore, in doing so, one loses the advantages that Rainbow provides, including architecture-level modeling and analysis, separation of adaptation concerns from system functionality, and flexibility to add and evolve self-adaptation capabilities.

Using development-activity data gathered from the Znn.com and TalkShoe examples, combined with anecdotal evidence from TalkShoe's chief architect and an exploratory analysis of hypothetical custom-solution scenarios, we can make qualitative comparison of effort between Rainbow-based versus custom-solution adaptation engineering. To prevent unfairly skewing the comparison in favor of Rainbow, we have made estimates that favor custom-solution activities wherever possible.

For custom-solution efforts, in terms of the four adaptation engineering stages described above, we have assumed that *domain analysis* required the same amount of time as with Rainbow, that *model capture* generally required zero time, that *design and implementation* required a minimum of one month on top of the Rainbow-based time for developing the monitoring and effecting capabilities, and that each *update and modification* required a quarter of the design/implementation time to complete because of buried and dispersed adaptation logic.

We treat the TalkShoe example as a best-case scenario due to its relatively straightforward adaptation function; for the Rainbow-based data, we recorded one day on domain analysis that was not included in the TalkShoe development-activity data. We treat the Znn.com system as an

average-case scenario due to its correspondence with typical N-Tier IT systems; for the Rainbow-based data, we estimated about two weeks of domain analysis. We acquire Rainbow-based time values by rounding up actual development-activity data to the nearest whole number of work days or weeks, whichever is closer. While we do not have concrete data for a worst-case scenario, we can reasonably estimate worst-case Rainbow-based development time to be an order-of-magnitude worse than the average case.

Table 7.1 summarizes the task-based estimation of effort comparing Rainbow against custom solution. In the worst-case scenario, a Rainbow instantiation has no reusable style, gauges, probes, and effectors from prior efforts. The adaptation engineer must construct many elements from scratch, so the primary advantage of Rainbow comes from the reusable framework. Thus, Rainbow-based initial development does no better in the worst case than custom solution. According to this coarse-grained task estimation of efforts, excluding the worst case, Rainbow yields effort savings of 2–5× over custom solution for initial development of self-adaptation capabilities. Thereafter, Rainbow achieves additional savings of up to two orders-of-magnitude (6–192×) over custom solution when evolving self-adaptation capabilities. The upfront effort of engineering the Rainbow generic framework accounts for the time-saving over custom solution.

Table 7.1: Task-based estimation of effort: Rainbow versus custom-solution

(h/d/w/m/y, 1w=5d, 1m=22d)	**Best case (TalkShoe)**		**Average case (Znn.com)**		***Worst case****	
Stage (Development)	Custom	Rainbow	Custom	Rainbow	Custom	Rainbow
Domain analysis (D)	*1d*	1d	*2w*	*2w*	*5m*	*5m*
Model capture (M)	-	4h	-	2d	-	*1m*
Design/implementation (I)	*1m 4d*	4d	*1m 2w*	2w	*6m*	*5m*
Development total:	**~1.25m**	**~0.25m**	**~2m**	**~1m**	**~1y**	**~1y**
Update/modification (U*)	*6d*	0.25h	*8d*	4h	*6.5w*	*1w*

*: italicized items are estimates

7.3 Claim: Transparency

As noted in Section 1.4.1, we evaluate **transparency** by demonstrating that our self-adaptation framework provides

- Constructs to specify strategies for different dimensions;
- Support to combine those dimensions meaningfully; and
- Mechanisms to effect adaptation that integrate strategies to achieve multiple objectives.

We further demonstrate the expressiveness of the language by

- Interviewing system administrators to understand the administrative process; and
- Qualitatively assessing how well Stitch represents the adaptation class of sys-admin tasks.

Transparency of adaptation decisions entails being able to (a) understand the adaptation process, (b) compose adaptation actions, and (c) automate adaptation choice. To understand the adaptation process, an approach must *hoist* to *first-class* the self-adaptation concepts of modeling,

monitoring, decision-making, and effecting changes; it must also provide capabilities that make these concepts concretely accessible to engineering. To compose adaptation actions, an approach must be able to harness knowledge of adaptations from domain experts of different domains separately, since an expert might not know enough of another domain to consider the related issues, and then allow compositions of these adaptations, possibly in a different context, to achieve overall objectives. To automate adaptation choice, an approach must provide the means of determining when an action should be selected and how utility profiles and preferences affect the resulting action chosen.

System administrative tasks can be broadly categorized into three classes: software installation and deployment, system configuration and maintenance, and fire-fighting or system adaptation. The first class of administrative tasks, though potentially automatable, usually requires connecting hardware, manually moving physical media, etc.; we do not consider self-adaptation support for this class. In contrast, the third class is precisely the class for which we want to provide self-adaptation support, while the second class is only sometimes automatable. Therefore, we assess expressiveness of the self-adaptation language by how well we can represent self-adaptation scripts for the adaptation-class of administrative tasks.

The customization of Rainbow for Znn.com demonstrates all three aspects of transparency. As illustrated in Section 5.1, Rainbow provides framework capabilities for distinct aspects of the adaptation process: models, probes, gauges, and effectors; Rainbow also provides representation constructs—operators, tactics, strategies, and utility dimensions and profiles—to help concretely capture steps of adaptation, value systems, and preferences for trade-off decisions.

Znn.com demonstrates strategies that capture adaptation for separate dimensions, in particular, a strategy to fix response time by varying content fidelity, a strategy to recover fidelity level, and a strategy to reduce the cost of under-utilized servers. Using utility profiles and preferences, the strategies can be composed across 4 partially conflicting objectives to achieve high system utility. The Znn.com experiment shows that Rainbow's utility theory-based algorithm to integrate the strategies works *effectively*.

Finally, the use of utility theory to compose strategies enables automation of adaptation choice. As already illustrated in Section 4.5, varying the utility preferences allows the adaptation engineer to affect which strategy is selected. In fact, if a simulation harness is used (as we did for Znn.com), the adaptation engineer could further explore what-if scenarios to determine the desired input of utility profile and preferences. In short, Znn.com demonstrates that utility profiles and preferences provide an explicit means for adaptation engineer to analyze, account for, and resolve conflicts in adaptation objectives to automate choice.

Our experiences with Chief Architect Bob Pawlowski at TalkShoe provide another piece of external supporting evidence that the Rainbow framework makes self-adaptation concepts *understandable*. During our initial conversations about Rainbow customization for TalkShoe scenarios, it was straightforward to establish as baseline the adaptation concepts of architecture models, probes, gauges, effectors, objectives, conditions, operators, tactics, and strategies. Due to their natural match with intuition, these concepts allowed Bob and me to make design progress effortlessly; they are then directly implementable using the Rainbow framework. In short, this anecdotal data point demonstrates that Rainbow makes self-adaptation concepts *understandable*, and thus accessible, to an engineer wanting to add self-adaptation capabilities to his system.

Three pieces of evidence from sys-admins support the expressiveness of Stitch for representing system-administrative tasks in the adaptation category.

Almossawi's interview of White offers an initial data point, revealing that a sys-admin behaves with implicit goals or objectives (*adaptation objectives*), reacts to system problem indications (*adaptation conditions*), takes specific actions (*tactics*) to resolve problems given observed conditions (*strategy*: condition-action-delay), considers the impact of actions against implicit criteria (*cost-benefit attributes*), and desires monitoring and early alert capabilities. In other words, as shown in parentheses, the interview data substantiate the core Stitch concepts. Note that although Almossawi has defined arbitrary numbers for three objectives, the key observation is that objectives and cost-benefit are concepts integral to the sys-admin task. The interview further reveals that sys-admins need the automation help that Rainbow can provide. As illustrated, the Stitch script derived from Almossawi's analysis of the interview sufficiently and naturally captures the system adaptation scenario described by the sys-admin.

The two system adaptation scenarios originating from Almossawi's personal experience offer a second data point. The bare-bone listings of the tasks in Section 6.6.3 reveal a few important elements in support of the concepts that Stitch embodies: the implicit *adaptation objectives* of keeping the webserver in normal working order; *adaptation conditions* like "high server load"; specific adaptation actions (*tactics*), for example, "restart httpd process"; and condition-action-delay patterns (*strategies*) evident in Almossawi's task sequence, particularly observations of system conditions, such as "check disk space". Note that Almossawi further identifies decision criteria for choosing actions, which supports the Stitch concept of tactic *cost-benefit attributes*.

The system adaptation knowledge embodied in the *netbwe* Perl scripts offers a third data point. As illustrated by the details in Section 6.7, we can identify from the *netbwe* source code adaptation concepts supported by the Rainbow framework, and in particular, the Stitch language. The adaptation objectives are separately inferred from conversations with the sys-admin, but that fits our expectation, where implicit business objectives have to be elicited from system owners to enable self-adaptation. All other elements are found in the Perl module, often buried innocuously as a single line of source code. For example, line 133 of the Netbwe::API::violations subroutine (listed under D.2 on page 198),

```
if ( $mach->{$key}{Rate} <= $limit) {
```

captures an important adaptation condition where a machine has reached a bandwidth quota limit. While the limit value is at least *not* hardcoded (it is defined elsewhere in a table of bandwidth quota categories), burying this logic deep in a subroutine greatly hinders the ability to understand and analyze adaptation decisions. In contrast, the EscalateViolationStates strategy listed on page 123 captures the violation condition prominently with the hasViolations predicate. As evident from this exercise, Stitch allows codifying and expressing adaptation concerns and decisions in an explicit, straightforward, and natural form.

In short, evidence from sys-admins supports both the validity of self-adaptation concepts designed in Stitch and the expressiveness of the language for representing adaptation logic.

7.4 Summary

In this chapter, we addressed how the examples and evidence collectively fulfill the thesis claims of **generality**, **cost-effectiveness**, and **transparency**, summarized in Table 7.2.

Table 7.2: Summary of example evidence toward thesis evaluation

Claim	**CSSys**	**Libra**	**UnivSys**	**TalkShoe**	**Znn.com**	**SysAdm**	**Netbwe**	**All**
General — Rainbow applies to many *styles* and multiple *objectives*?								$\surd$
- 3+ styles	(CS)	(SvcC)	(CS)	(N-tier)	(N-tier)	(SvcC)	(SvcC)	4
- 3+ objectives	(*perf*)	(*perf*+*cost*)	(*security*)	(*avail.*)	(4)	(*bw*+*avail.*)	(3)	5
Cost-effective — Rainbow demonstrates *reuse* (between instances) and *ease of use*?								$\surd$
- Reusable	$\surd$		$\times$	$\surd$		$\times$	$\surd\times$	$\surd$
- Easy to use	$\times$	$\times$	$\times$	$\surd$ 93h	$\surd$ 34h	$\times$	$\surd\times$	$\surd$
Transparent — Rainbow makes adaptation process explicit and understandable								$\surd$
- Understand.	$\surd-$	$\surd-$	$\surd-$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
- Composable	$\times$ 1D	$\surd-$	$\times$ 1D	$\times$ 1D	$\surd$	$\surd$	$\surd$	$\surd$
- Analyzable	$\times$ 1D	$\surd-$	$\times$ 1D	$\times$ 1D	$\surd$	$\surd\times$	$\surd\times$	$\surd$

$\times$: not applicable, thus not demonstrated
$\surd\times$: possibly demonstrated with more in-depth study
$\surd-$: partially, but was not fully designed or supported

Chapter 8

Discussion of Issues and Limitations

In this chapter, we discuss potential limitations of our approach and reflect on our main design choices. Specifically, we address the following issues:

- *Central* control as an important design choice for Rainbow;
- *Asynchronous* adaptation interactions and its ramifications to *uncertainty* and *failure*;
- Issues with *closed-loop feedback* control;
- Limitations of Stitch's *expressiveness* to represent certain system administrative concepts;
- Limitations to adaptation inherent with the use of a *model*;
- Limitations to automating adaptation choice using *utility theory*; and
- Impact of *framework*, *reusable* artifacts, and adaptation *experience* on *cost-effectiveness.*

8.1 Central Control

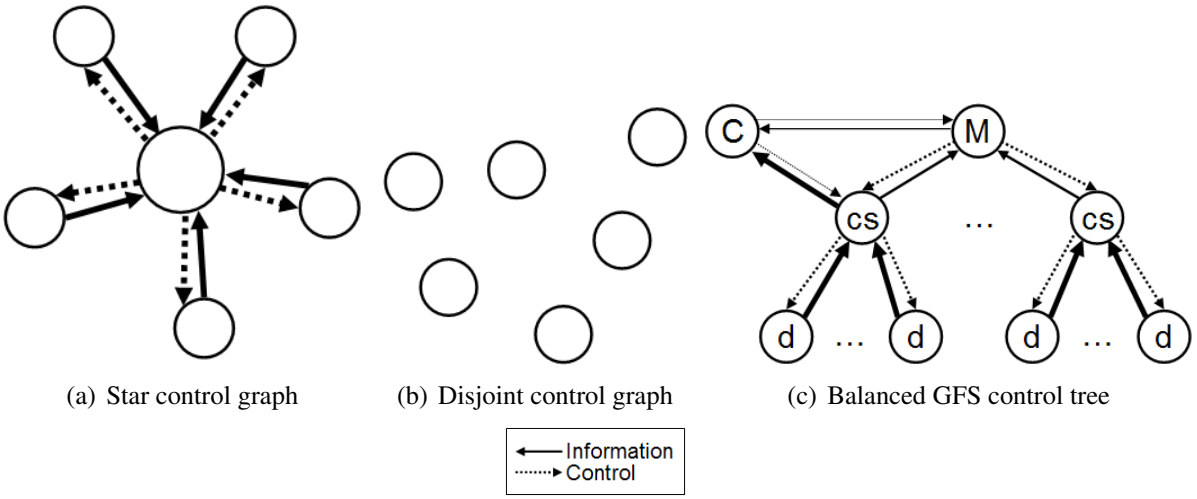
We have chosen central control for the Rainbow approach. Centralizing control allows a single point for decision-making, creating a powerful control mechanism while simplifying the adaptation process. Combined with the use of an architecture model (discussed below in Section 8.5), central control provides global perspective of the system, allowing adaptation decisions that takes end-to-end system conditions into consideration. On the other hand, central control introduces a single point-of-failure and leads to potential scalability and efficiency issues. An alternative approach is to distribute control across the individual mechanisms.

For control systems, the degree of control generally ranges from despotic and centralized to highly federated and distributed. As Section 2.2 (control paradigms) indicates, the degree of control depends on the availability of information and of control knobs. We can view control systems abstractly as directed graphs, where each node controls, or is controlled by, another node, and an edge is either an information or a control flow. This abstraction allows us to characterize control systems using five attributes:

- Nodes: (1) entity *autonomy* as a node property, ranging from *none* to *full*; (2) count of *controllers*, or nodes with outgoing control edges, ranging from 1 to N

- Edges: (3) count of *information* edges (availability of information) and (4) count of *control* edges (availability of control knobs); thickness indicates amount of data flow
- Configurations: (5) Graph type, ranging from fully disjoint (no information nor control edges), to complete (nodes fully connected by information and control edges), to hierarchical, such as a tree (root node as top controller) or a forest (multiple root nodes)

Varying these 5 attributes produces different kinds of control systems with corresponding trade-offs, such as, (a) power of the controllers, (b) susceptibility to failure, (c) scalability, and (d) efficiency. Controller count and entity autonomy both determine the extent to which the system is susceptible to controller failure.



(a) Star control graph (b) Disjoint control graph (c) Balanced GFS control tree

Figure 8.1: Control graphs for representative points

At one extreme is in an autocratic system, abstractly a star graph (Figure 8.1a), where all information edges flow to one controller node and all control edges originate from that same node. The controller node has complete power to control the behavior of the system, but is also a single point-of-failure. Not only does controller failure cause the whole system to fail, but control information is most likely irrecoverable. Such a system would also be difficult to scale and is likely to be inefficient unless the information edges are thin. In contrast, with a fully distributed system such as an intelligent swarm, abstractly a disjoint graph (Figure 8.1b), many nodes can fail without severely crippling the overall system, but the behavior of the system is emergent and may be difficult to predict [BDT99].

An important midpoint, as exemplified by the Google File System [GGL03], is a tree configuration (Figure 8.1c) with two tiers of control. In each GFS cluster (see Figure 8.2), a master controller is the root node, chunkservers are the second-level nodes, and disks are leaf nodes. A master controller controls many chunkservers, but information edges from them are thin. Chunkservers connect to many disks, and while information edges from disks are thicker, chunkserver replication provides redundancy and increases efficiency. Clients (assume that these nodes are transient on the graph) interact only briefly with the master, but reads data from chunkservers directly. Chunkservers have some autonomy to control and access data from disks,

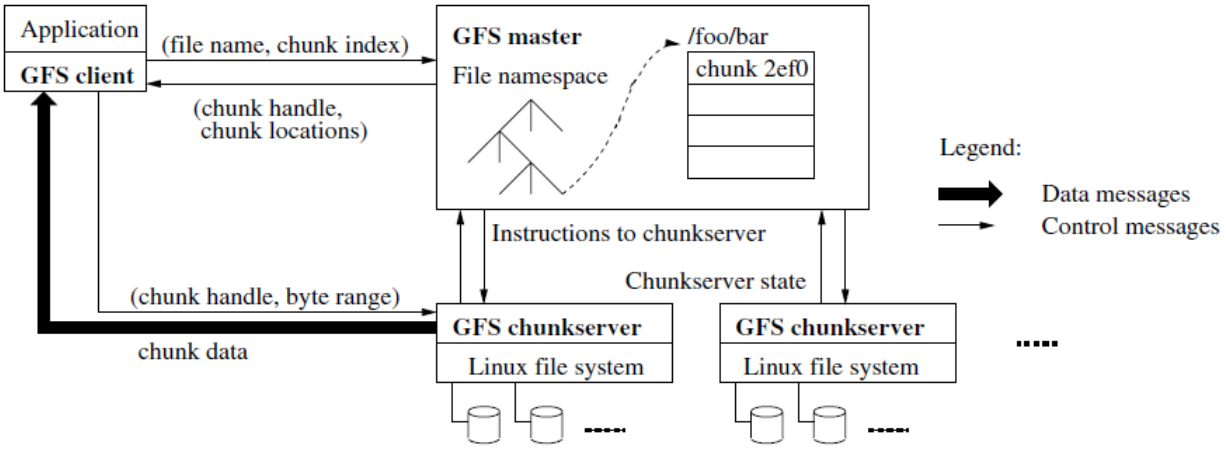


Figure 8.2: GFS Architecture (extracted from [GGL03])

but the master has global knowledge and manages namespace, creates content, replicates and re-replicates content, and rebalances replicas.

Although the master is a single point-of-failure, it restarts quickly, or an external entity could quickly replace the master with another node to bring the cluster back online. Thus, GFS has a powerful controller, designed for frequent failures, that recovers quickly and scales well and efficiently to serve file requests. GFS clearly illustrates that a central, powerful controller does not necessarily indicate a show-stopping, single point-of-failure, or performance bottleneck.

Although we adopt primarily a central control for Rainbow, we have also designed mechanisms to distribute control responsibilities, making Rainbow another midpoint that strikes a similar balance as GFS. Abstractly a tree configuration (Figure 8.3), the Rainbow controller box marked "Architecture Level" (see Figure 3.2 on page 29) is the root node, Rainbow Delegates, which manage probes and effectors in distributed target system nodes (see Section 5.1.1), are the second-level nodes; probes and effectors are the third-level nodes; and target system entities that interact with the probes and effectors are leaf nodes. Information flows from target system entities to probes to the controller; control flows from the controller to effectors to target system entities. The controller has global, end-to-end knowledge of the target system, allowing it to make and enact well-informed adaptations.

The infrastructures of Rainbow have been designed to distribute responsibility while coping with failures at several levels. Delegates allow probe and effector components to be deployed on distributed computing nodes. Depending on the event infrastructure used, the communication infrastructure for Rainbow's components can be configured with varying levels of message delivery guarantees. Delegates can restart probes and effectors when they fail. Likewise, a daemon can respawn delegates that fail.

The Rainbow controller appears to be the single point-of-failure. Within the Rainbow controller are gauges and operators, elided from our discussion so far. Data flow from probes to gauges, and control flow from operators to effectors. The Architecture Evaluator, Adaptation Manager, and Strategy Executor each store minimal internal states. These components can all be restarted upon failure without much loss of control capabilities. The Model Manager is the locus of information flow from the gauges. If it fails, Rainbow ceases to function. Fortunately,

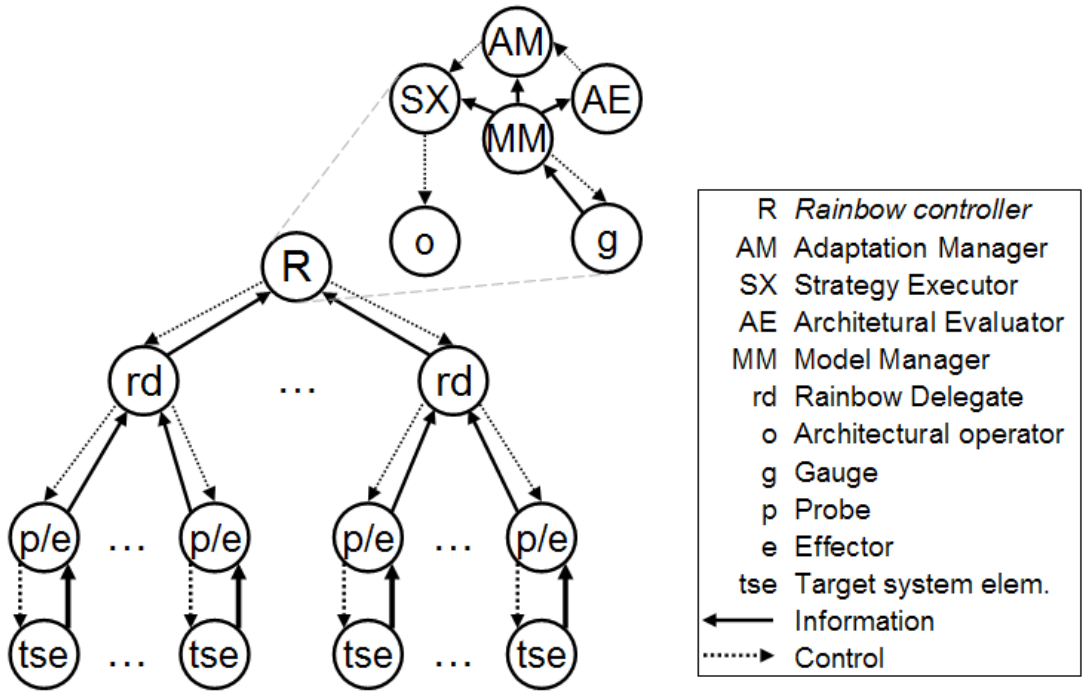


Figure 8.3: Control graph for Rainbow

as discussed in future work below (Section 9.2.2), we have techniques for quick model recovery.

With slightly more engineering, the Rainbow controller can be restarted upon failure, or an external entity could quickly respawn the controller on a different computing node if the controller node fails completely. Therefore, from the root node down the control hierarchy, Rainbow components may fail, and may even be susceptible to a single point-of-failure; however, our design enables distribution of responsibility and respawning of components at various levels, making Rainbow resilient to most forms of (component) failures.

Due to the large amount of monitoring information potentially required in an architecture-based approach, scalability and efficiency are important design concerns. Since probes and gauges are deployable on distributed nodes, with the appropriate choice of event infrastructure, they do not pose scalability and efficiency issues. The communication and computation bottleneck resides with the Model Manager, but we have already made a few mitigating design decisions, including update-triggered (versus periodic) evaluation of model constraints. Performance overhead in Znn.com experiment runs indicate that there appears to be considerable room for improvements to Rainbow's performance.

Unfortunately, the adaptation time window of Rainbow does limit its domains of applicability. From probes, to gauges, to the model, to adaptation decision-making, to operators, and back down to effectors, the inherent time delay in the adaptation cycle is, according to performance data observed in this thesis, hundreds of milliseconds at best, seconds on average, and minutes at worst. This time scale limits our ability to apply Rainbow to most embedded and real-time systems, which usually operate at sub-millisecond time-scale. In addition, the computational resources required to maintain the model—memory footprint on the order of tens of

megabytes—prevents the use of Rainbow on highly resource-constrained devices, such as handhelds and embedded devices. Fortunately, probes and effectors can be designed to operate at a much shorter time-scale using much less memory, so if the Rainbow controller can operate on a separate, more powerful computing node, and if the millisecond-to-second time-scale fits, it is still possible to apply Rainbow on systems with constrained resources.

8.2 Asynchronous Interaction and Uncertainty

By design, Rainbow's adaptation mechanisms interact asynchronously with the target system. This design reflects our adoption of a control systems approach to self-adaptation. This design decouples the controller from the target system, making the self-adaptation infrastructures reusable over different styles of system. On the other hand, asynchrony and loose coupling introduce monitoring and action uncertainties between the target system and the controller. In particular, there is greater uncertainty in knowing when a problem has been detected and when a change has been successfully effected. In an alternative, synchronous approach, or one where the adaptation logic is part of the target system implementation (e.g., via exception-handling), problem detection can be more direct and adaptation changes can be effected immediately.

We discuss specific framework design choices that allow us to balance the needs for asynchrony while addressing uncertainties. In this discussion, we shall distinguish the mechanisms of adaptation interactions from the implementation-level mechanisms, where interactions are often realized with call-return constructs, e.g., method calls. By adaptation interactions, we mean the activities that occur in the adaptation cycle: monitoring, detection, decision, and action.

We are interested in systems that continue to operate in the face of problems, where the systems are not taken offline while adaptations are performed. Hence, adaptation mechanisms must interact asynchronously with the target system. In the Rainbow framework, monitoring events occur asynchronously at the probe and gauge levels. The gauge consumer thread in the Model Manager updates the architecture model asynchronously, without blocking for model evaluation, and vice versa. The Evaluator triggers the Adaptation Manager asynchronously, so that the Evaluator is not blocked by the adaptation process. The Adaptation Manager passes the selected strategy to the Executor asynchronously. Although the Executor carries out tactics synchronously by blocking until an operator (translated to an effector) completes, it does *not* block for changes to take effect in the system. Instead, the strategy defines a timing delay for the Executor to observe the expected effect in the system (as reflected in the model).

The primary disadvantage of asynchronous, concurrent interactions are issues of race conditions and deadlocks. Therefore, we simplify the design for the Model Manager, the Evaluator, the Adaptation Manager, and the Executor by choosing synchronous interactions within each component. We therefore constrain the potential sources of concurrency problems to a small set of interactions between these key components, which we carefully manage. The primary source of race conditions would arise during model update, so our implementation observes the rule that all model updates occur only from within the Model Manager thread. At the implementation level, the points where one adaptation component invokes another are the "critical sections" where deadlocks can arise, so here we minimize computation and constrain side-effects to setting (in Java parlance) a synchronized member variable.

Having discussed our design choices to address uncertainties from asynchrony at the mechanism-level, we now discuss uncertainties within the phases of self-adaptation. In designing the Rainbow framework, we have made assumptions about these adaptation uncertainties to mitigate design complexities.

1. *Monitoring*: to get accurate measurement, we assume well-designed probes
2. *Detection*: to determine if problem exists, we assume that it suffices to detect symptoms
3. *Decision*: to decide on an adaptation action, we assume utility preferences can be elicited
4. *Action*: to carry out adaptation actions, we assume sufficient probe coverage to observe system state and regroup from failure

Adaptation decision (#3) has received central focus in this thesis. When deciding on the course of adaptation remedy, not only must numerous factors be considered, choices must also be made between both similar kinds of actions as well as actions that have potentially juxtaposing effects. As we have shown, utility theory, combined with a stochastic model for the action outcome, enable trade-off comparison despite uncertainties. We consider the difficulties of eliciting utility preferences in Section 8.6.

For problem detection (#2), a constraint violation identifies an opportunity for adaptation. Given more than one constraint, multiple violations may occur in one adaptation round. To handle this case, two approaches are possible. The first approach prioritizes the constraint violations and resolves the problem of highest impact (in effect, highest utility), particularly useful for sys-admins who have limited time and cognitive resources to respond to multiple problems at once. The second approach evaluates and chooses from a known list of potential actions to assess which delivers the highest utility when applied to the present system conditions. This approach eliminates the need to prioritize, and it is most suitable when the set of actions (i.e., strategies) are available upfront and the assessment can be done in a reasonably short time.

In Rainbow, we have taken the second approach since the mechanism has a known list of strategies, and the duration required to analyze and select the strategy that delivers the highest utility is less than a fraction of a second. Together with an overall adaptation cycle on the order of a second, the resolution time of Rainbow is much shorter than typical human resolution time for system problems, from minutes to hours or longer [Wik08f]. We look at an alternative to constraint-based detection in Section 9.2.2 (future work).

Rainbow's adaptation decision mechanism depends on the monitoring infrastructure (#1) to deliver proper readings. Thus far we have assumed that system probes are well-designed and appropriately deployed to provide accurate readings. However, a combination of framework mechanisms and careful engineering can eliminate reliance on this assumption. Most of the mechanisms are already provided by Rainbow, but enhancements are possible with days to weeks of effort. Making monitoring work entails proper attention to the following issues:

- Core functionality: The adaptation integrator should ensure that probes and gauges process input, compute results, and report output as expected. Rainbow has provided interfaces and abstract classes, as well as sample probes and gauges, to help focus the engineer on the core functionality.
- Event delivery: The adaptation integrator should ensure probes and gauges are properly connected, and events are delivered to the right place. Rainbow implements basic probe

and gauge infrastructures to facilitate event delivery.

- Timing: The adaptation integrator should consider how often probes and gauges report values, from and to which nodes information propagates, capacity of links carrying report values, and when and how events propagate up the monitoring chain. Rainbow's top-level customization of gauges and probes provides the engineer a complete picture and facilitates spotting missing links.

The primary uncertainties in the action step (#4) are whether the effector completed its operations and whether it achieved its intended effects. In either case, as long as there is sufficient probe coverage to observe the current target-system state, the natural cycles of self-adaptation will handle failure recovery.

8.3 Closed-Loop Feedback Control

The Rainbow framework is, at its core, a closed-loop feedback-control system. Closed-loop feedback control enables a continuous cycle of reacting to target-system problems and making adjustments. However, Rainbow is potentially susceptible to similar control issues as those typically encountered in control systems. Hellerstein, et al., defines four principle properties of concern for feedback control of computing systems: stability, accuracy, settling time, and overshoot (SASO) [HDPT04]. Our framework design incorporates some of these control concepts.

We can define stability in terms of the property of *hysteresis* for physical materials with magnetic characteristics: a system exhibits hysteresis if it retards the effect of changes in forces acting upon it (adapted from Webster's definition). In other words, a stable system is one that reduces or dampens the effect of perturbations upon it. Conversely, a system is unstable if minor perturbations results in oscillation of system behavior. Using the Znn.com example, if a rise in visitor traffic causes the controller to oscillate between executing a strategy that reduces fidelity to achieve better average response time, and one that increases fidelity to achieve better average content quality, then the controller is unstable.

Hellerstein, et al., describe techniques to design controllers for discrete computing systems, specifically by relating measurable outputs to controllable inputs, deriving equations to model the system, and designing controllers that ensure SASO properties based on the system model. The design techniques they describe are particularly suitable for systems where one can establish mathematical equations to relate inputs to outputs. In contrast, Rainbow's target systems generally do not fit this profile; nonetheless, the SASO properties are useful concepts for achieving stable Rainbow control. So far, we have incorporated the concepts of accuracy, settling time, and overshoot, to a limited extent.

In Rainbow, *accuracy* of control is achieved by the chain of constructs—adaptation conditions, strategy conditions of applicability and condition-action pairs, and tactic conditions of applicability—that ensure the appropriate action is chosen and executed for the observed system symptoms. We apply *settling time* in the form of the time-window of delay for observing the effect of an executed tactic. One disadvantage is that settling time requires manual input by the adaptation engineer, but we have made it an explicit customization point in the framework. Finally, overshoot can occur when an action has an impact of control beyond what is necessary;

thus, the concept of *overshoot* manifests itself in the scope of control impact of each tactic. To illustrate overshoot, consider from the Znn.com example an earlier version of the tactic raiseFidelity, which raises the fidelity level of *all* servers by a given number of steps, rather than just the servers with the lowest fidelity:

```
1  // An earlier version that causes overshoot:
2  tactic raiseFidelityOvershot (int step, float fracReq) {
3    condition { ... }
4    action {
5      // first find the set of servers with room to raise fidelity
6      set servers = { select s : T.ServerT in M.components | s.fidelity <= M.
           MAX_FIDELITY_LEVEL - step};
7      for (T.ServerT s : servers) {
8        S.setFidelity(s, java.lang.Math.min(s.fidelity+step, M.MAX_FIDELITY_LEVEL));
9      }
10   }
11   effect { ... }
12 }
13
14 // Current version that does not overshoot:
15 tactic raiseFidelity (int step, float fracReq) {
16   condition { ... }
17   action {
18     // first find the lowest fidelity set
19     set servers = { select s : T.ServerT in M.components | s.fidelity <= M.
           MAX_FIDELITY_LEVEL - step};
20     int lowestFidelity = M.MAX_FIDELITY_LEVEL;
21     for (T.ServerT s : servers) {
22       if (s.fidelity < lowestFidelity) {
23         lowestFidelity = s.fidelity;
24       }
25     }
26     // find only servers with this lowest fidelity setting, and raise fidelity
27     servers = {select s:T.ServerT in M.components | s.fidelity <= lowestFidelity};
28     for (T.ServerT s : servers) {
29       S.setFidelity(s, java.lang.Math.min(s.fidelity+step, M.MAX_FIDELITY_LEVEL));
30     }
31   }
32   effect { ... }
33 }
```

When the strategy ImproveOverallFidelity countered the effect of strategy SmarterReduceResponseTime, the earlier version of tactic raiseFidelity caused an overshoot of control and resulted in unstable oscillation between the two strategies. Therefore, it is possible to reduce overshoot by reducing the control impact of a tactic.

While the Rainbow approach embodies feedback control, Rainbow could also be enhanced with proactive capabilities. The current design of the Architecture Evaluator evaluates the architecture model as triggered by model updates. One possible design enhancement would be to add periodic, global model evaluation. Although potentially more resource-intensive, if combined with predictive information about the system [PGS+07, Che08b, Pol08], periodic evaluation could allow Rainbow to find opportunities for improvement in an anticipatory, proactive manner.

8.4 Stitch Expressiveness: Operator, Tactic, and Strategy

By observing commonly performed system administration tasks, we have extracted a set of three constructs—operator, tactic, and strategy—and thus a basic ontology of an adaptation language

for automating mundane tasks in system management. Together, the concepts of adaptation conditions, tactic cost-benefit attributes, strategy conditions of applicability, quality dimensions, utility preferences, and strategy selection form an adaptation language with the expressiveness to represent human adaptation expertise and the flexibility to incorporate dynamic preferences.

It is worth discussing why three constructs are needed: one might argue, would two not suffice, one for the primitive steps, and one for the predefined plan? The need for three is motivated by abstraction, packaging, and separation of concerns. The operator, as provided by an architectural style, embodies element modification within that style, while, as mapped to an effector, it embodies specific actions to effect changes in the target system. In other words, the operator provides the crucial reuse link between the adaptation mechanism and the architecture model as well as the translation link between the adaptation mechanism and the target system.

However, the operator does not suffice as an adaptation primitive for two reasons. First, the style writer who provides the architectural operators generally cannot know the adaptation context in which the operators will be used. Knowledge of the adaptation context and, particularly, the impact of operators on the utility dimensions for the target system are separate concerns of the adaptation engineer. Second, the adaptation engineer may need to define larger steps of adaptation than is provided by architectural operators. Hence, we need a second, distinct concept. As already stated in Section 4.3.5, the third concept, strategy, embodies explicit decision choices of the sys-admin and provides a packaging construct to constrain adaptation to individual domains of expertise for tractable reasoning.

Having discussed the need for three constructs, we now discuss which can invoke which. The invocation relationship as currently implemented in Stitch is summarized in Table 8.1.

Since the operator is not specified in Stitch, Stitch does not designate whether operators can call other operators. If realized as programs, operators might conceivably invoke another operator as a matter of clarity and reuse. However, operators can neither invoke tactics nor strategies. In Stitch, a tactic may only invoke an operator, not another tactic nor strategy. Allowing the tactic to invoke a strategy would simply not make sense for the adaptation semantics we desire. As already stated in Section 4.3.4, nesting tactics makes cycles possible, complicating the single-step semantics of the tactic as well as the evaluation of the condition and effect blocks. In particular, it is not obvious what role the condition block of a called tactic should play: should it be allowed to abort the calling tactic? should it be ignored altogether?

By design, a strategy may only invoke a tactic and not an operator, for reasons already stated for distinguishing the tactic from the operator. As already mentioned in Section 4.3.5, whether a strategy should be allowed to invoke another strategy is not entirely a settled point. If such invocation were allowed, the obvious semantics would be to graft the tree of the invoked strategy

Table 8.1: Invocation relationship between Stitch operational constructs

	Operator	Tactic	Strategy
Operator	?	X	X
Tactic	$\surd$	X	X
Strategy	X	$\surd$	X

at the node of the invoking strategy. However, we have chosen not to allow such invocation for simplicity and to prevent confounding domains of expertise (packaging).

Finally, we discuss strategy versus a generated plan. Given the current state of the target system as observed through the model (conditions) and a set of available tactics (actions), it is possible to use a planning algorithm to search for the best sequence (or path) of tactics, and thus generate a strategy. In our approach, we elicit strategies rather than use a planning algorithm to find the best adaptation path for a number of reasons. Planning is ideal for exploring a large space of potential paths, but in the domains we are targeting, adaptation decisions are often known or, at least, constrained in scope. Because plans are generated on-the-fly, from a utility perspective, the uncertainty and potentially large number of generated plans make it difficult to perform an end-to-end, closed analysis of adaptation outcomes.

In contrast, during adaptation, the set of strategies to explore is much smaller and, thus, a more efficient set on which to perform utility-based analysis of adaptation outcomes. Strategy yields consistent, verifiable, and reusable adaptation outcomes. Furthermore, the notion of *strategy* is intuitive, giving sys-admins greater control over Stitch-type representations and its decision outcome; the planning alternative would not be at all intuitive.

8.5 Limitations to Adaptation Using a Model

The Rainbow approach uses architecture and environment models for self-adaptation. Among its many benefits, an abstract model allows the adaptation engineer to focus on important properties of the target system, to capture end-to-end conditions of the target system, and to perform upfront analysis. However, using an abstract model raises questions surrounding what cannot be represented at the model level. Specifically, does the model granularity allow capturing all measurable properties, for example, socket connection queue length that might be useful to adapt against buffer overflow? Does the architecture model allow dealing with non-quantifiable or non-measurable properties? And, how does one deal with scope issues for the environment model?

Although the architecture model captures *abstract* structure and properties of the target system, its main purpose is to allow architects to capture the properties they *care* to model [CBB+03]. In that sense, model granularity does not preclude capturing properties as low (in terms of proximity in the application stack to the operating system) as socket connection queue length, if that is the property of adaptation concern for a particular target system.

System attributes that are typically not quantifiable are still possible to model in the architecture, as we have already demonstrated in Section 6.3 by capturing security risks using relative risk index [But02]. Similarly, this sort of modeling is done commonly in system design, where engineers often have to make trade-offs between system attributes that are subjective and not easily quantified. In such cases, engineers are often able to impose quality judgment (e.g., attribute a_1 is more important than a_2) and rank attributes (cf. ATAM utility scenario analysis [KKC00]), which are then translatable to quantifiable properties (e.g., $[r_1, r_2, \cdots] = [0.4, 0.3, \cdots]$). In short, one can often derive a quantitative, ranking-based property from a qualitative system attribute to enable system self-adaptation for that quality attribute.

One of the important purposes for modeling is to manage scope, to focus resources and attention on aspects of the system that are important. For the environment model, it would not

be feasible nor useful to model the entire Internet of computing nodes. However, the question of scope is easily settled by focusing on the target system in question. The environment resources directly accessed by the target system would tend to be the important environment elements to model and monitor, hence, our use of the notion of *hop count* (cf., Section 3.3.1).

The issue of what can be modeled leads to a larger issue of how general Rainbow is to different systems: based on the data points so far, is Rainbow self-adaptation generalizable to many quality attributes of concern? This issue is best addressed by pointing out a key feature of the Rainbow framework. Applying the generic adaptation capabilities of the framework to a particular quality attribute depends on whether Rainbow can (a) model it, (b) probe it, and (c) effect some target system changes for it. If the answer is *yes* to all three parts, then Rainbow is applicable to that quality attribute. To probe information about either the target system or environment, in addition to advances in probing technology, systems currently provide support for extracting system states in the operating system, such as the /proc/ for system and I/O statuses in Linux kernel (version 2.5 and newer). Similarly for effecting changes, standard operating system level control mechanisms exist, such as the SysV daemon control architecture.

8.6 Limitations to Using Utility Theory

We have chosen to leverage utility theory to automate strategy selection, but utility-based selection is only a partial solution to achieving trade-off decision making. The problem of reconciling conflicting quality attributes is known to be a fundamentally hard problem. Quality attribute trade-off is thus an essential (vs accidental) problem that cannot simply be automated. However, features of an essential problem can be explicitly identified to facilitate finding a solution. In our approach, we provide explicit mechanisms that make features of the quality trade-off problem prominent to facilitate automation of trade-off decision making:

1. We rely on humans to define the value system (utility preferences) and the actions (strategies and tactics) to build up self-adaptation capabilities
2. We rely on humans to relate impact of actions to value dimensions
3. We provide the mechanisms that automate the trade-off process and select the best (as defined by user utility) strategy once these inputs are provided

The explicit specification of objective attributes and enumeration of preferences and relative weights over those attributes not only allow fine-grained control over selection outcomes, but also provide reasoning about selection decision via a quantitative framework. Evaluation using user utility and preferences over outcomes enables a rigorous selection of the strategy that achieves the best trade-off across multiple dimensions.

A final concern is the seemingly large amount of information to elicit from experts and owners of the managed system: the utility profiles, preference weights, cost-benefit attributes, and probabilities. We observe that sys-admins already have to process a large amount of information when making decisions. Our approach simplifies the sys-admins' job by giving structure to the large quantity of information, providing placeholders for them in our framework, and allowing the information to be supplied incrementally to achieve the desired self-adaptation capabilities.

8.7 Framework, Reuse, and Experience on Cost-Effectiveness

The Rainbow framework is instrumental to saving cost and effort in engineering a system for self-adaptation. Because the framework is designed to be generic, investment to develop generic infrastructure functionalities may be deemed expensive, but such investment should be considered amortized cost across new instantiations of the Rainbow framework.

Using the Rainbow framework, there are two levels of reuse that contribute to cost-savings:

1. First-order reuse of the generic and common infrastructures;
2. Second-order reuse of customization artifacts: probes, gauges, effectors, tactics, strategies.

In general, second-order reuse of artifacts from one target system to another depends not only on the size of the reuse library, but also on similarities between the two systems: the more similar the system styles and quality concerns, the more artifacts can be reused. Specifically, probes and effectors depend on assumptions about the implementation platform of the target system, and probes and gauges monitor a particular quality attribute. However, to reuse tactics and strategies from one target system to the next, the minimum requirement is only that the system styles be similar. In such cases, reusing a tactic would require, at worst, (a) changing the name of the import architectural style and operator library at the script level and (b) rewriting the cost-benefit attributes for the set of utility dimensions in the new system. Reusing a strategy would require, at worst, (a) changing the name of the imported architectural style, (b) making sure all imported tactics are available, and (c) adjusting timing delay for observing tactic effects.

Although in both cases of TalkShoe and Znn.com, the Rainbow customization effort matched closely with the best-case effort estimation in Table 7.1, the customization in both cases was done by an expert user of the Rainbow approach and framework. In contrast, we anticipate a new adopter of Rainbow to expend either the average- or worse-case effort, while climbing the initial learning curve, depending on the complexity of the self-adaptation capabilities being added to the target system. However, as the adaptation engineer gains experience, we anticipate the customization effort to approach the best-case in subsequent adaptation instances.

8.8 Summary

In this chapter, we have addressed a number of potential issues and limitations with the Rainbow approach. Rainbow strikes an important balance of combining a centralized controller with a set of distributed infrastructures and a design that is resilient to failures at various levels of the system. While the interaction between Rainbow's adaptation phases are asynchronous to allow concurrent activities, the resulting sources of uncertainty can be mitigated with design assumptions and well-engineered mechanisms. The Rainbow framework applies techniques from control theory to achieve stable control. Most major limitations that we have identified can be mitigated: Stitch's expressiveness, the use of an abstract model, and achieving trade-off using utility. Finally, the cost-effectiveness of the approach is dependent on three factors: framework reuse and cost amortization, artifact reuse, and engineer's experience in using the framework.

Chapter 9

Conclusion and Future Work

In this final chapter, we enumerate the thesis contributions and discuss research issues for future work to improve the capabilities of Rainbow in particular, and our understanding of self-adaptive systems in general. We conclude with a summary of this thesis.

9.1 Thesis Contributions

This thesis advances the state-of-art in software engineering by improving our understanding of the approaches, mechanisms, and trade-offs for software architecture-based self-adaptation, specifically with the following contributions:

1. Characterization of
   (a) a general *approach* to architecture-based self-adaptation, including what constitutes generic, reusable functionalities and what particular aspects must be customized for a class of systems and a set of quality dimensions;
   (b) a *language* that models system administration concepts, represents system adaptation knowledge, enables domain expertise to be captured independently, and facilitates analysis of adaptation strategy integration across domains of expertise; and
   (c) an *adaptation engineering* software *process* to add and evolve self-adaptation capabilities for existing systems, leveraging the approach developed in this thesis;
2. Demonstration of particular *techniques* to enable self-adaptation, including a utility-based algorithm to compose and integrate independent adaptation strategies from different quality dimensions, the use of an MDP framework to model the adaptation process, and the adoption of control theory concepts for adaptation control;
3. A packaged *tool* set that integrates the self-adaptation approach, provides a subset of techniques, and supports the adaptation engineering process; and
4. Demonstration of coverage of a representative region of the self-adaptation system space, helping to understand (1) what styles of system are amenable to adaptations, (2) what domains of concerns can be addressed by automatic control, and (3) what kinds of system adaptation are feasible.

9.2 Future Work

In this section, we discuss a few self-adaptation capabilities that are not currently realized by Rainbow, but that are logical next steps. We also discuss a few important research issues to further enrich our understanding of self-adaptive systems and advance the body of knowledge in software engineering. Specifically, we discuss:

- Short-term framework improvements, including tool support;
- Improving framework fault-tolerance;
- Alternatives to constraint-based detection of opportunities for adaptation;
- Potential of the environment model and how it can be enriched;
- Enhancement of adaptation with control system techniques and learning;
- Design choices to enable *preemption;*
- Improving trust and and confidence in self-adaptation;
- Incorporating user input to improve relevance of system adaptation;
- Other potential uses of Rainbow; and
- Research issues beyond Rainbow.

9.2.1 Short-Term Framework Improvements

For future work within the next three to six months, we can make these framework improvements:

- Using Java concurrency library to facilitate scheduling and management of the multi-threaded Rainbow runtime components;
- Adding support for linear, sigmoid, and other common utility functions for Utility Profile specification;
- Adding error-handling mechanisms in the Strategy Executor to handle action errors identified in Section 8.2: translator-mapping, effector-connection, and effector-execution;
- When a Stitch script accesses an architecture model instance, restricting references to generic collections of model elements of a certain type (e.g., M.components of type T.ServerT), and never to specific model instances (e.g., M.admin0);
- Adding a two-staged response when Rainbow encounters system symptoms for which it does not find a strategy: first, Rainbow alerts a human administrator when the unresolvable symptom is initially encountered; second, Rainbow invokes a worst-case strategy for severe cases where an unresolvable symptom reflects an unstable target system; and
- Optimizing Rainbow performance, including increasing the efficiency of communication within the monitoring mechanisms, batching gauge updates into coarser time slots, improving incremental model evaluation, possibly distributing the model among multiple machines for a really complex architecture.

Moreover, we can increase the adoptability of Rainbow by improving tool support, as we encountered during the TalkShoe case study. The Rainbow framework is intended as a tool to enable

adaptation engineering. To that end, working with Bob Pawlowski has provided one positive data point that another engineer could use the Rainbow framework to engineer self-adaptation for his or her own IT infrastructure. However, we identify a few items of future work to improve the framework's ease of use for software engineers:

- Rainbow customization requires numerous configuration files, which are separate for good reasons, including separation of concerns, division of responsibility, and maintainability and flexibility. As part of the Rainbow Adaptation Integrated Development Environment (as proposed in Section 5.2 on page 84), an intelligent customization GUI (e.g., with wizards) could further increase the efficiency of managing these configuration files and provide added benefits, including templated configuration elements, specialized forms for utility description, and default or recommended values.
- To encourage greater use of Stitch scripts and prevent *stitching* frustration, the tooling support for Stitch should be improved with features such as editor integration with the architecture model, model-aware as well as context-aware auto-completion, typechecking feedback, and runtime awareness as well as error-reporting of missing Rainbow pieces.
- The packaging of the Rainbow SDK should allow the Rainbow libraries and codebase to be effortlessly integrated into the development environment of the target system. In particular, we should separate the Rainbow APIs for standalone probes and effectors, which may need to integrate with target system code.
- Shell scripts could be provided for common deployment tasks.

The Rainbow deployment architecture presents an issue for nodes with critical resources. One of the simplifying assumptions made during the design of the Rainbow runtime deployment architecture is that each host in the target system would instantiate a RainbowDelegate (a Java process with a memory footprint of up to 32 MB). Effectors would execute from the local RainbowDelegate process, eliminating the need for elaborate, cross-machine effector communication protocols. However, system owners would most likely not want to deploy a RainbowDelegate on host machines with critical processes or constrained resources. For such cases, it may make sense to provide lightweight versions of the delegate process or other lightweight effector mechanisms.

9.2.2 Medium-Term Rainbow Research Issues

For future work within the next six months to a year, we can address a number of research issues to enhance the Rainbow approach.

Fault-Tolerant Schemes To increase the fault-tolerance of Rainbow's mechanisms, as discussed in Section 8.1, we need to ensure quick model recovery. There are a number of standard techniques, including persistence and checkpointing. The architecture model, managed using the Acme library, can be persisted to file periodically, and the internal model reconstructed from file within seconds. The disadvantage of this technique is the potential inconsistency in states between the reconstructed model and the target system. The alternative, upon Rainbow recovery, is to recreate the architecture model by querying the present system state, with the downside of

taking more time to reconstruct the model. A third technique is to checkpoint the model periodically, log updates to the model at redundant points located separately from the Model Manager, and then, upon Rainbow recovery, recreate the model by combining the check-point and log data.

Detecting Opportunities for Adaptation For coarse-grained but straightforward detection of potential system problems, we have favored rapid recognition using a few key variables, over complicated analysis, for greater efficiency and effectiveness [Gla06]. We have used a push-based, model update-triggered, constraint-based detection to determine when the target system requires adaptation. However, another design choice would be to use an envelope of *qualities-of-service* to assess the need for adaptation in a pull-based manner. As illustrated in Figure 5.5 on page 80, a more sophisticated QoS Analyzer could replace the Simple Constraint Evaluator to continuously evaluate the system for opportunities of adaptation based on qualities of service. This design alternative would advance the capabilities of Rainbow toward a self-healing system with softer adaptation *precisions*, as described by Shaw [Sha02].

The disadvantage is to incur greater computation overhead and potentially cause more disruptions to the target system due to more frequent adaptations and fine-tuning. However, one mitigation is to account for disruption as a quality dimension [PSGS04]. Next, if we can reduce the computational cost and time delay of the adaptation roundtrip to something reasonably small, say milliseconds, then a QoS-based adaptation would become favorable.

Enriching the Environment Model In this thesis, we have taken first steps in using an explicit environment model to enable self-adaptation. However, further research is required to investigate what environment information to model for various quality dimensions, explore and enrich the notion of an environment *style*, and characterize the relationship between the environment and the architecture.

In the Znn.com instantiation, we explored a performance-oriented environment model. Our technique is to treat the environment as equal in status to the architecture and thus to apply the same modeling approach, including using the constructs of *style*, *type-instance*, *property*, and even *operator*, and modeling environment-specific components, connectors, and interfaces. For mechanism, as described in Section 3.3.1, the Model Manager maintains the environment model, updating its properties via readings from environment resource gauges. Finally, the environment model is accessible from strategies and tactics using the same model query mechanism as the architecture model.

An important area of future work, applicable to ubiquitous and mobile computing where the user's environmental context provides crucial information for adaptation, is the ability to adapt the environment, which extends beyond contextual systems to human users. Similar in process to engineering for target system adaptation, environment adaptation would require resolving three problems: describing the environment style supplied (by a *system context engineer*); developing probes to acquire environment information, which may be challenging if information is distributed and the entities and resources monitored are not in one's control domain; and developing effectors to control environment entities and resources. In a related approach, Poladian has developed models, mechanisms, and algorithms to optimize the use of environment resources to achieve a certain user purpose [Pol08].

Enhancing Adaptation Control Utility-based strategy selection is one of several possible methods of realizing adaptation, which could be implemented as "plug-ins" to change the behavior of the *Adaptation Manager*. These levels differ from one another in terms of control timing, knowledge of the past, and knowledge of the predicted future. Utility-based selection may be considered a first-cut, direct approach. We have also taken the logical next step to incorporate simple history in subsequent strategy selections, for example, to avoid repeating bad actions (cf., Section 4.3.5). Once one has access to history, one can integrate learning as part of the selection process to avoid oscillation and to improve selection quality. The next leap is to integrate predictions into the monitoring infrastructure, possibly via predictor gauges, to anticipate certain quality-of-service problems, such as an anticipated rise in CPU load, drop in available bandwidth, or even change in the state of user tasks [PGS+07, Pol08].

An orthogonal path to improve the Adaptation Manager is a multi-layered approach to develop mechanisms on top of Rainbow to monitor Rainbow's adaptation history, learn patterns, and dynamically update (a) tactic cost-benefit attributes, (b) strategy branch probabilities, and (c) utility profile (which requires incorporating user input). In the further future, one can apply advanced planning-based strategy selection to improve automation, similar to the robotics architecture, e.g., MDS. In addition, one can explore extended adaptation capabilities, including proactive adaptation to optimize conditions, exploratory adaptation to try different strategies to determine the best, and, finally, continual and homeostatic adaptations [Sha02].

Preemption Problem preemption may be necessary when a problem arises that is severe enough (e.g., security intrusion) to interrupt an adaptation process already in progress. For such a scenario, we would introduce the notion of *priority* or *severity* to architectural constraints. When the Architecture Evaluator detects a constraint violation, it could then determine, by comparing problem priorities, whether to interrupt the adaptation process presently executing in the Adaptation Manager. The Evaluator would have the power to preempt an adaptation, but this capability requires addressing three issues: (1) Who captures priority, where? (2) Which constraint is the current strategy adapting for? (3) How should unfinished adaptation be interrupted cleanly?

The *adaptation integrator*, with some level of knowledge across the adaptation domains, most appropriately has the knowledge to capture and encode problem priority. Assigning levels of severity to Acme design rules may be implemented in one of two ways, as an enhancement to the Acme design rule construct that allows different levels of *invariant*, or with a meta-property *level* for rules. The integrator may then assign levels to design rules to capture constraint priority.

The key concern in interrupting an in-progress adaptation is whether the target system would be left in a consistent state. If the Adaptation Manager is in the middle of an adaptation process, it is most likely evaluating a strategy. Since we have required the tactic to execute as a single adaptation step, it follows that the interruption of an adaptation should *not* occur in the midst of tactic execution. A natural interruption point is at a branch point of a strategy; alternatively, the *strategy writer* could be allowed to explicitly identify interruptible branches, but the exact mechanism would require additional design.

Since Rainbow's strategy-selection process does not consider the set of violated constraints, it is not obvious which constraint is being handled by the selected strategy. One possibility is to use the strategy's conditions of applicability to relate strategies to constraints. Another possibility

is not to worry about this relationship, but to assume that the utility dimensions and preferences already capture the preemptive condition. In this case, when a severe-priority constraint is violated, the Adaptation Manager can be requested to perform only the strategy-selection process to determine if the currently-executing strategy should be preempted.

Trust and Confidence in Self-Adaptation Adoption of the Rainbow approach is positive evidence of its usefulness and effectiveness. One major hurdle to adoption is user trust in self-adaptation, where transparency of the decision and adaptation process is crucial. Capabilities of the framework should be enhanced to report adaptation actions, especially any failure of adaptation, to the user. We can gain trust by gradually building user confidence in the decision and adaptation accuracy of the framework. An effective approach is to ask the user initially for confirmation on decision steps, then allow the user to gradually assign greater control to the adaptation mechanism. The adaptation mechanism can also be designed to learn the good versus bad decisions from interactions with the user.

Incorporating User Input and Task-Oriented Computing In the Rainbow approach, the system owner and users play an important role in the customization and engineering phase for self-adaptation, but we have not addressed issues of ongoing changes in user needs once Rainbow is deployed and operating. In other words, there is a parallel distinction to traditional software systems between the design and the runtime phases of Rainbow, which can be removed in future work; the key lies in allowing dynamic updates of the customization content. Most of the customization points that require business input, such as the strategy metadata mentioned above, are potential points for dynamic update, and thus for incorporating user input.

In our current implementation, we have assumed the strategy repertoire to be static: it is provided once and does not change after Rainbow deployment. Since our strategy selection algorithm allows the Adaptation Manager to consider different sets of strategies with each round of adaptation, the strategy repertoire could be dynamically updated as necessary after Rainbow is deployed and running. With additional engineering effort, Rainbow can be extended to allow such dynamic repertoire update. In addition to modifying the repertoire, given an existing strategy, its meta-information could conceivably be modified dynamically. In relation to strategies, the branch probabilities, time window of tactic-effect observation, cost-benefit attribute values, and utility profiles are all data that can be updated dynamically. Such dynamic update allows us to incorporate user input to adjust or improve the accuracy of the adaptation decision process.

However, it may not always be feasible or desirable to expose these customization points to the user directly. Instead, inputs to these customization points may result from user tasks that have been explicitly modeled and captured, as in the Aura architecture [SG02]. When user tasks change, their utilities and preferences will likely change for the system's quality of service. A component that bridges between Rainbow and a user task management component such as Prism [Sou05] would provide dynamic adaptation parameters using user task models.

Potential Uses for the Rainbow Approach From the TalkShoe interaction, we observe two new potential applications for Rainbow, one for system administration, and the other to aide system deployment. A system management challenge is to provide the sys-admin with sufficient

information of the system states, preferably in one location, to help make appropriate decisions. Rainbow's central console could facilitate management by showing system states through the architecture and environment models updated live with the target system states. For this purpose, we need to perform a systematic user interface design catered to sys-admin needs. For system deployment, we have learned that, despite configuration tools like the Spring framework, performing the coordinated startup of software components distributed across different machines remains an error-prone chore. In future work, Rainbow could be applied to help manage such versioned, distributed software deployment, perhaps by modeling an architecture model where software versions are component attributes, version dependencies are connector attributes, and version compliance are composition constraints.

9.2.3 Longer-Term Research Beyond Rainbow

Looking back at software engineering issues, we draw insight from our work that might help guide software engineers in designing future systems, so that system self-adaptations can increasingly be achieved in a cost-effective and transparent manner.

Software engineers should design systems for runtime self-adaptation capabilities, including providing explicit hooks for probing system state and events and for effecting changes, such as to start, stop, restart system components or to update component states.

Software engineers should design systems for dynamism and adaptability, with conscious effort to *hoist* points of system variability, make communication interactions and assumptions explicit, support system changes as transaction and allow rollback and recovery, and make individual components restartable and possibly swappable.

Finally, software engineering research should continue finding answers to the question: How can we systematically analyze the behavior of adaptive system and assure certain system properties? Researchers have been advancing research on this front, but so far, we cannot answer this question well. However, we are interested in specific formalism 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 [KG06].
- How does the semantics of adaptation interact with the semantics of the architecture? The primary research issue here is to determine what properties we could prove.
- Can we automatically determine timing delay of an adaptation operation? The challenge is to formalize *effectors* to enable timing analysis.
- How can we use a Markov Decision Process to generate or solve for a strategy that does X? The hard problem is to come up with novel ways of handling the raw state space of the system, which, even in the simplest of cases, has an infinite number of states. Once we can manage states, then it becomes possible to model adaptive systems in MDP and solve for the optimal policy, Π, to determine the best strategy of tactics from any given state.

The Rainbow approach embodies feedback control, and its framework components correspond to the elements of a feedback control system (Figure 1.2 on page 4) as illustrated in Fig-

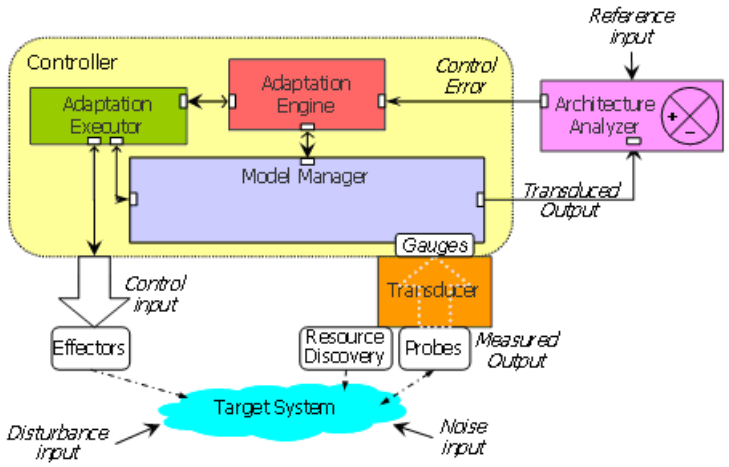


Figure 9.1: Rainbow illustrated as a feedback control system

ure 9.1. The control-system quality attributes of the Rainbow framework would correspond to the SASO design points and the system model would be composed of the target system's architecture model. In future work, we could potentially create self-adaptation system-design tools to aid adaptation engineers in analyzing system inputs and outputs to establish mathematical relationships, or in incorporating analysis from control system design tools such as MatLab Control Toolbox or SimuLink.

Further beyond, Rainbow is potentially an initial step toward goal-based computing, where achievable goals are the primary input to the computer, and the computer automatically does what is needed to attain the goal. One of the main features of Rainbow is allowing an adaptation engineer to define a value system, define a set of adaptation strategies, and then relate the strategies to the value system to achieve objectives. These three elements form the basic ingredients for goal-based computing. The challenge is to determine how to generate these elements automatically from a set of goals. The first step is to determine patterns and rules of connection between a goal and a value system, an action, and the action effects. Goal-based computing is not far-fetched; example designs exist that partially realizes it, for instance, the Mission Data System architecture envisioned by researchers at the NASA Jet Propulsion Laboratory, which consists of goal networks and goal elaboration capabilities to carry out goals through controllers, estimators, sensors, and actuators [Gat00, DIMG07]. Similar approaches exist for modern robotics systems.

9.3 Summary

This thesis set out to demonstrate the effectiveness of the Rainbow architecture-based approach to system self-adaptation. Modern systems grow increasingly complex while needing to adapt to changes: user preferences may change, environment resources may vary, dependent software components may disappear or fail. Architecture-based self-adaptation hoists properties of concern into prominent architecture models and enables a target system to be monitored and adapted at run time.

In Chapter 1 we posited the thesis that *we can provide software engineers the ability to add*

and evolve self-adaptation capabilities cost-effectively for a wide range of software systems and for multiple objectives by defining a self-adaptation framework that factors out common adaptation mechanisms and provides explicit customization points to tailor self-adaptation capabilities for particular classes of systems and quality objectives. This thesis led to the claims of generality (to a broad spectrum of styles for common quality dimensions with which modern architects are concerned), cost-effectiveness (relative to existing specialized solutions, to engineer and evolve self-adaptive systems), and transparency (making the adaptation process understandable, actions composable, and choice automatable for routine system adaptations).

In Chapter 2, we surveyed areas of related work and identified contributing disciplines—software architecture, control theory, utility theory—as well as limitation of related approaches, which demonstrate point solutions for particular classes of systems and fixed quality dimensions, but lack a self-adaptation framework generally applicable to different styles and multiple quality objectives, the ability to represent administrator adaptation concepts as explicit and operational entities, the mechanism to automatically decide the best course of adaptation, and an integrated approach that saves engineers time and effort.

In this thesis we overcame the limitations in the state-of-art by providing an architecture-based approach to system self-adaptation. We started from the thesis claims as requirements in Chapter 3. The overall Rainbow approach provided a framework of self-adaptation mechanisms to enable architecture-based monitoring and adaptation, a language for expressing and composing self-adaptation objectives and strategies, and a process for incrementally engineering self-adaptation capabilities. We presented the features and semantics of the Stitch self-adaptation language in Chapter 4, and detailed the customization points of the reusable framework infrastructures in Chapter 5. To demonstrate that we fulfilled our requirements and thus satisfied the thesis claims, we presented seven concrete pieces of evidence in Chapter 6 and evaluated them in Chapter 7. In summary, this thesis fulfills the requirements as follows.

Generality: the Rainbow approach leverages the notion of software architectural style to characterize and define explicit customization points for tailoring common, reusable infrastructures of the framework to specific styles for multiple quality concerns. Evidence: 5 example systems and 2 scenarios, which together demonstrate coverage for

- 3 architectural styles: client-server, service-coalition, N-tier;
- 5 quality concerns: *performance*, *cost*, *content fidelity*, *availability*, *security*.

Cost-effectiveness: the Rainbow approach prescribes an adaptation engineering process that guides a systematic, incremental customization of Rainbow to target systems. Evidence:

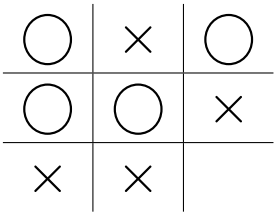
- Two instances of reuse from CSSys to Libra, and between TalkShoe and Znn.com;
- TalkShoe as external evidence of ease in customizing Rainbow;
- Task-based estimation of self-adaptation engineering effort with development data from TalkShoe (34 h) and Znn.com (93 h).

Transparency: the Rainbow approach provides a self-adaptation language that separates the concerns of self-adaptation into distinct concepts to facilitate reasoning, provides explicit constructs to represent adaptation knowledge separately for different domain expertise, and enables integrating adaptation strategies from the different domains by quantitatively selecting the adaptation strategy that achieves the best trade-off satisfying multiple objectives. Evidence:

- Sys-admin examples substantiate the expressiveness of Stitch's self-adaptation concepts;
- Znn.com and (in part) TalkShoe demonstrate that Rainbow makes self-adaptation understandable, composable, and automatable.

Closing Remarks This thesis presented evidence in support of Rainbow's generality to architectural styles and quality dimensions, cost-effectiveness via reusable artifacts and ease-of-customization, and transparency in understanding, composing, and automating adaptation choices. By defining and designing an architecture-based self-adaptation framework with common, reusable adaptation mechanisms that are customizable to specific styles of systems for multiple quality objectives, we have provided software engineers the ability and tools to add and evolve self-adaptation capabilities cost-effectively to a broad spectrum of systems for typical qualities of concern.

To complete evaluation of this thesis, please solve the following puzzle:



Bibliography

[AAG93] Gregory D. Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993. 2.1

[AAG95] 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. 1.2, 3.1.1

[ACM02] ACM. Adaptive middleware. *Communications of the ACM*, 45(6), June 2002. 3.3.2, 9.3

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proc. of the 24th International Conf. on Software Engineering*, pages 187–197, New York, NY, 2002. ACM. 2.3.1

[Agh02] Gul A. Agha. Introduction. In *Communications of the ACM* [ACM02], pages 30–32. 2.3.1

[All97] Robert J. Allen. *A Formal Approach to Software Architectures*. PhD thesis, Carnegie Mellon University School of Computer Science, May 1997. 2.1, 3.1.1

[Alm06] Ali Almossawi. Personal and email communications, May–August 2006. Summer independent study, carried out interview design and process, shared insight on system administration. 6.6

[AVCL02] 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*, pages 203–210. ACM Press, 2002. 4

[Bak04] D. Robert Baker. A decision table based methodology for the analysis of complex conditional actions. *Methods & Tools: Global knowledge source for software development professionals – Fall 2004*, 12(3):23–35, 2004. 2.2

[Bal01] Robert Balzer. Probe run-time infrastructure. `http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt`, 2001. 2.3.1, 3.3.2

[Bat00] John A. Bather. *Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions*. John Wiley & Sons, Baffins Lane, Chichester, West Sussex PO19 1UD, England, first edition, July 13, 2000. 1.3.1, 2.2

[BBKS05] Felix Bachmann, Len Bass, Mark Klein, and Charles Shelton. Designing software architectures to achieve quality attribute requirements. *Software, IEE Proceedings*, 152(4):153–165, August 5, 2005. 2.1

[BCDW04] 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*, pages 28–33, New York, NY, 2004. ACM. 2.3.3

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley Longman, 1998. 1.2, 2.1

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley Longman, second edition, 2003. 2.1

[BDH03] Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003. 1.1.1

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford University Press, Inc., New York, NY, 1999. 3(a) on p. 272. 8.1

[Bis06] Bill Bishop. Personal communications, May 2006. Discussed (a) possible case study for Rainbow and (b) system administrative task. 6.6

[BJ03] Len Bass and Bonnie John. Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software*, 66(3):187–197, June 15, 2003. 2.1

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, volume 3527 of *LNCS*, pages 1–17. Springer, June 13–14, 2005. 2.3.3

[BK07] Karun N. Biyani and Sandeep S. Kulkarni. Mixed-mode adaptation in distributed systems: A case study. In *Proc. of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, page 14, Washington, DC, 2007. IEEE Computer Society. 2.3.1

[BLKW97] M. Barbacci, T. Longstaff, M. Klein, and C. Weinstock. Quality attributes. Technical Report CMU/SEI-96-TR-036, Software Engineering Institute, 1997. 2.1, 7.1

[BMD+03] David J. Bruemmer, Julie L. Marble, Donald D. Dudenhoeffer, Matthew O. Anderson, and Mark D. McKay. Mixed-initiative control for remote characterization of hazardous environments. In *36th Hawaii International Conf. on System Sciences (HICSS-36 '03), CD-ROM / Abstracts Proc.*, page 127. IEEE Computer Society, January 6–9, 2003. 2.2

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, 1996. Layers for Information Systems, p. 47. 7.1

[But02] Shawn A. Butler. Security attribute evaluation method: a cost-benefit approach. In *Proc. of the 24th International Conf. on Software engineering*, pages 232–240. ACM Press, May 19–25, 2002. 2.1, 6.3, 7.1, 8.5

[CBB+03] 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. 1, 3.3.1, 8.5

[CdlFBS01] 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*, pages 134–140, New York, NY, 2001. ACM. 2.3.3

[CDS01] *Proc. of the Working Conf. on Complex and Dynamic Systems Architecture*, December 12–14, 2001. 1.3.1, 2.3.1, 3.3.2, 9.3

[CGS+02a] Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, and Peter Steenkiste. Using architectural style as a basis for self-repair. In Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, *Software Architecture: System Design, Development, and Maintenance*, pages 45–59, Massachusetts, USA, August 25–30, 2002. Kluwer Academic Publishers. 6, 6.1

[CGS+02b] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for grid computing. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 389–398, July 23–26, 2002. 6.1, 6.1

[Che08a] Owen Cheng. Personal communications, April 2008. No Easter in April. :o). 6.6

[Che08b] 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. 8.3

[CHG+04] 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. 3.3.2, 3.3.2, 6, 6.2

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, Dec 2004*, volume cs.OS/0406005, December 2004. 2.2, 2.3.1

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proc. of the 8th International Workshop on Software Specification and Design*, page 16. IEEE Computer Society Press, 1996. 2.1

[CMU08] CMU Architecture Based Language and Environments. Acme - The Acme Architectural Description Language and Design Environment. `http://acme.able.`

cs.cmu.edu/, 2008. 2.1, 5.1.2

[CN01] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. The SEI Series in Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 3rev ed edition, August 30, 2001. 7.2

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001. 2.3.1

[CU01] Carnegie Mellon University and University of California, Irvine. xAcme. http://www.cs.cmu.edu/~acme/pub/xAcme/, 2001. 2.1

[Cur94] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994. 2.3.1

[CV02] Nathan Combs and Jeff Vagel. Adaptive mirroring of system of systems architectures. In Garlan et al. [GKW02], pages 96–98. 2.3.1

[DC01] 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*, pages 81–88, London, UK, 2001. Springer-Verlag. 2.3.3

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proc. of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag. 2.2

[DGK+01] Peter A. Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Dean Sutherland. The architecture of the remos system. In *Proc. 10th IEEE Symposium on High Performance Distributed Computing*, 2001. 6.1

[DHT01] 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. 1.2, 2.1

[DHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. [GKW02], pages 21–26. 1.3.3, 2.3.3

[Dij75] Edsger W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. 4.3.5

[DIMG07] Daniel D. Dvorak, Michel D. Ingham, J. Richard Morris, and John Gersh. Goal-based operations: An overview. In *Proc. of the Infotech@Aerospace Conf. and Exhibit*, May 7–10, 2007. 9.2.3

[DLSD01] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicodemos Damianou. A policy deployment model for the ponder language. In *Proc. 7th IFIP/IEEE International Symposium on Integrated Network Management (IM'2001)*. IEEE Press, May 2001. 2.2

[Dol00] Shlomi Dolev. *Self-Stabilization*. The MIT Press, Cambridge, MA, 2000. 2.2

[Dur94] John Durkin. *Expert Systems: Design and Development*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1994. 2.2

[FLV00] 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. 1.2

[FPS02] Jason Flinn, SoYoung Park, and Mahadev Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proc. of the 22nd International Conf. on Distributed Computing Systems (ICDCS'02)*, pages 217–226. IEEE Computer Society Press, July 02–05, 2002. 2.3.1

[Fry03] Colleen Frye. Self-healing systems. *Application Development Trends*, pages 29–34, September 2003. 1.1

[Gat00] Erann Gat. The mds autonomous control architecture. In *CD-ROM (ISOMA 9947) Proc. of the 4th World Automation Conf. (WAC '00)*, June 2000. 9.2.3

[GC03] A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003. 1.1, 2.3.2, 3.1.2, 3.3.2, 7.1

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004. 6, 6.1, 6.2, 7.2

[GCS03] 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, pages 61–89, New York, NY, USA, 2003. Springer-Verlag, Inc. 6, 6.1

[GGK+01] Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S. Kc, and Janak J. Parekh. An active events model for systems monitoring. In CDSA [CDS01]. 2.3.1

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003. (document), 1.1.1, 8.1, 8.2

[GKW02] David Garlan, Jeff Kramer, and Alexander Wolf, editors. *Proc. of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, New York, NY, USA, November 18–19, 2002. ACM Press. 1.2, 1.3.1, 9.3

[Gla06] Malcolm Gladwell. *Blink: The Power of Thinking Without Thinking*. Penguin, January 2006. 9.2.2

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organizing software architectures for distributed systems. In Garlan et al. [GKW02], pages 33–38. 2.2, 2.3.3

[GMW00] 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 Univ

Press, 2000. 1.2, 2.1, 3.3.3

[GR91] Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis. In *Proc. of the 13th International Conf. of Software Engineering*, pages 23–34, Los Alamitos, CA, USA, May 1991. IEEE Computer Society Press. 1.1.1, 1.3, 2.1, 2.3.3

[Gro02] Philip Gross. MEET. `http://www.psl.cs.columbia.edu/meet/index.html`, 2002. 2.3.1

[GSC01] David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In CDSA [CDS01]. 2.3.1, 3.3.2, 5.1.3

[GSRU07] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, 2007. 2.3.3

[HDPT04] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. 2.2, 8.3

[Hei98] George T. Heineman. A model for designing adaptable software components. In *COMPSAC '98: Proc. of the 22nd International Computer Software and Applications Conference*, pages 121–127, Washington, DC, 1998. IEEE Computer Society. 2.3.1

[Hew03] Hewlett-Packard Development Company. Adaptive enterprise. `http://h71028.www7.hp.com/enterprise/cache/6842-0-0-0-121.aspx`, 2003. 2.3.2

[How60] Ronald A. Howard. *Dynamic programming and Markov processes*. MIT Technology Press, June 1960. 2.2, 4.4.1

[HPUM07] 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*, page 15, Washington, DC, 2007. IEEE Computer Society. 2.3.3

[Hua04] An-Cheng Huang. *Building Self-configuring Services Using Service-specific Knowledge*. PhD thesis, Carnegie Mellon University, December 2004. 2.3.1

[IBM04] IBM. An architectural blueprint for autonomic computing, 2004. 1.1.2, 1.3.3

[IBM08] IBM developerWorks. Autonomic computing toolkit. `http://www.ibm.com/developerworks/autonomic/overview.html`, 2008. [Online; accessed 2-April-2008]. 2.3.2

[Int02] Intel Research Laboratory at Seattle. Plantcare. `http://seattleweb.intel-research.net/projects/plantcare/`, 2002. 2.3.2

[JBB03] J. Jann, L. M. Browning, and R. S. Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM Systems Journal*, 42(1):29–37, 2003. 2.3.1, 2.3.2

[Jur04] Mario Juric. Slashdotting of mjuric/universe. `http://www.astro.`

princeton.edu/~mjuric/universe/slashdotting/, January 13–15, 2004. Graph of actual Slashdot-effect traffic. 6.5.4

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. In *Communications of the ACM* [ACM02], pages 33–38. 2.3.1

[KG06] 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. 9.2.3

[KHW+01] 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. 1.1.1

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, CMU SEI, 2000. 8.5

[Le 98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998. 2.3.3

[LG07] 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)*, page 12, Washington, DC, 2007. IEEE Computer Society. 2.3.3

[LGSZ84] Edward D. Lazowska, G. Scott Graham, Kenneth Sevcik, and John Zahorjan. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, NJ, February 1984. 2.1

[LSZ+01] Joseph Loyall, Richard Schantz, John Zinky, Partha Pal, Richard Shapiro, Craig Rodrigues, Michael Atighetchi, David Karr, Jeanna M. Gossett, and Christopher D. Gill. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In *Proc. of the 21st International Conf. on Distributed Computing Systems*, pages 625–635, April 2001. 2.3.1

[MBO+07] Ronald Morrison, Dharini Balasubramaniam, Flávio Oquendo, Brian Warboys, and R. Mark Greenwood. An active architecture approach to dynamic systems coevolution. In *ECSA*, volume 4758 of *LNCS*, pages 2–10. Springer, September 24–26, 2007. 2.3.3

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, pages 137–153, Sitges, Spain, September 26, 1995. Springer-Verlag, Berlin. 1.3, 2.1

[MG04] Arun Mukhija and Martin Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 368–374, Washington, DC, 2004. IEEE Computer Society. 2.3.3

[Mic03] Microsoft Corporation. Dynamic systems initiative. http://www.microsoft.com/windowsserversystem/dsi/, 2003. 2.3.2, 3.3.2

[Mic08] Microsoft Corporation. System center operations manager 2007. http://www.microsoft.com/systemcenter/opsmgr/, 2008. 1.1

[Mit97] Tom Michael Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, New York, 1997. 1.3.1, 2.2

[MK96] 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*, pages 3–14, New York, NY, USA, 1996. ACM. 2.3.3, 2.3.3, 3.1.1

[MKMG97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, Jan–Feb 1997. 2.1

[MLR03] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003. 2.3.1, 2.3.2

[MM03] Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ESEC/FSE-11: Proc. of the 9th European Software Engineering Conf. held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 347–350, New York, NY, 2003. ACM. 2.3.1, 5

[Mon99] Robert Monroe. *Rapid Development of Custom Software Architecture Design Environments*. Technical Report CMU-CS-99-161, Carnegie Mellon University School of Computer Science, Pittsburgh, PA 15213, August 1999. 2.1, 3.3.4

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. volume 21 of *ACM Software Engineering Notes*, pages 24–32, New York, NY, USA, October 16–18, 1996. ACM Press. 2.1

[MPF+06] Marius Mikalsen, Nearchos Paspallis, Jacqueline Floch, Erlend Stav, George A. Papadopoulos, and Akis Chimaris. Distributed context management in a mobility and adaptation enabling middleware (madam). In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 733–734, New York, NY, 2006. ACM. 2.3.1

[MR97] Mark Moriconi and Robert A. Reimenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, SRI International, 1997. 2.1

[MRMM02] Marija Mikik-Rakic, Nikunj Mehta, and Nenad Medvidovic. Architectural style requirements for self-healing systems. In Garlan et al. [GKW02], pages 49–54. 2.3.1, 2.3.3

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Software Engineering: ESEC/FSE '97: 6th European Software Engineering Conference, held jointly with the 5th ACM*

SIGSOFT Symposium on the Foundations of Software Engineering, volume 1301 of *LNCS*, pages 60–76, New York, NY, USA, September 22–25, 1997. Springer-Verlag, Inc. 2.1

[NMMS99] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, pages 62–68, July 1999. 2.3.1

[NMMS02] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering*, Spring 2002. 2.3.1

[Obj01] Object Management Group. *OMG Unified Modeling Language Specification, v. 1.4*, September 2001. 2.1

[OGT+99] 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. 1.1.1, 1.1.1, 2.3.3

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *20th International Conference of Software Engineering*, pages 177–186, Los Alamitos, CA, USA, April 19–25, 1998. IEEE, IEEE Computer Society Press. 1.3

[Ore00] Peyman Oreizy. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine, 2000. 2.3.3

[PGM97] Robert H. Perry, Don W. Green, and James O. Maloney. *Perry's Chemical Engineers' Handbook*. McGraw-Hill, New York, NY, USA, seventh edition, 1997. 1.2, 2.2

[PGS+07] Vahe Poladian, David Garlan, Mary Shaw, Bradley Schmerl, Joao Pedro Sousa, and Mahadev Satyanarayanan. Leveraging resource prediction for anticipatory dynamic configuration. In *Proc. of the 1st IEEE International Conf. on Self-Adaptive and Self-Organizing Systems (SASO '07)*, pages 214–223, July 8–11, 2007. 2.2, 8.3, 9.2.2

[Pol08] 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. 8.3, 9.2.2, 9.2.2

[PSGS04] Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic configuration of resource-aware services. In *Proc. of the 26th International Conf. on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004. 9.2.2

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. 2.1

[PW02] Paul Pazandak and David Wells. Probemeister: Distributed runtime software instrumentation. `http://www.objs.com/ProbeMeister/paper/020523-probemeister.pdf`, May 23, 2002. Object Services & Consulting, Inc. Baltimore, MD. 2.3.1

[Ram02] Jay Ramachandran. *Designing security architecture solutions*. John Wiley & Sons, Inc., New York, NY, 2002. 7.1

[Rho08] Stephen Rhoton. Personal communication, January 8, 2008. Discussed example adaptive scripts (user bandwidth restriction) in CMU network administraton. 6.6, 6.7

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. 2.2

[San06] Greg Sandoval. Christmas shopping crush stalls walmart.com. `http://news.zdnet.co.uk/internet/0,1000000097,39284866,00.htm`, November 27, 2006. CNET News.com. 6.5.1

[SC81] Charles H. Sauer and K. Mani Chandy. *Computer systems performance modeling*. Advances in Computing Science & Technology. Prentice-Hall, Englewood Cliffs, NJ, March 1981. 2.1

[Sch06] Evan Schuman. Black friday turns servers dark at wal-mart, macys. `http://storefrontbacktalk.com/story/112406blackfriday.php`, November 25, 2006. StorefrontBacktalk - Techniques, Tools and Trades about Retail Technology and E-commerce. 6.5.1

[Sch08] Bradley Schmerl. AcmeStudio. `http://acme.able.cs.cmu.edu/AcmeStudio/`, 2001–2008. 2.1

[Sco02] Karyl Scott. Computer, heal thyself. *Information Week*, April 1, 2002. 1.1

[SEM89] Dale E. Seborg, Thomas F. Edgar, and Duncan A. Mellichamp. *Process Dynamics and Control*. Wiley Series in Chemical Engineering. John Wiley & Sons, New York, NY, USA, 1989. 1.1.1, 1.1.1, 1.2, 2.2

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996. 1.2, 2.1, 7.1

[SG98] 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. 2.1, 4

[SG02] Joao Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, *Software Architecture: System Design, Development, and Maintenance (Proc. of the 3rd Working IEEE/IFIP Conf. on Software Architecture)*, pages 29–43. Kluwer Academic Publishers, August 25–31, 2002. 2.2, 9.2.2

[SG04] Bradley Schmerl and David Garlan. AcmeStudio: Supporting style-centered architecture development. In *ICSE 2004*, 2004. 2.1, 5.1

[Sha95] Mary Shaw. Beyond objects: A software design paradigm based on process control. *Software Engineering Notes*, 20(1):27–38, January 1995.

[Sha02] Mary Shaw. "Self-Healing": Softening precision to avoid brittleness. In Garlan et al. [GKW02], pages 111–113. 2.2, 9.2.2, 9.2.2

[SL02] Morris Sloman and Emil Lupu. Security and management policy specification. *IEEE Network*, 16(2):10–19, March 2002. 2.2

[SL06] Alexandre Sztajnberg and Orlando Loques. Describing and deploying self-adaptive applications. In *Proc. 1st Latin American Autonomic Computing Symposium*, July 14–20, 2006. 2.3.3

[SN01] Mahadev Satyanarayanan and Dushyanth Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7(6):601–607, 2001. 2.3.1

[Sou05] 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. 9.2.2

[Spi05] Bridget Spitznagel. *Compositional Transformation of Software Connectors*. PhD thesis, Carnegie Mellon Univeristy School of Computer Science Technical Report CMU-CS-04-128, 2005. 2.3.1

[Sun02] Sun Microsystems. The sun fire(tm) v1280 server architecture. `http://www.sun.com/servers/midrange/pdfs/V1280_wp_final.pdf`, November 2002. See "Introducing N1" on p 40. 2.3.2

[Tai06] Paul Tait. Irwin's death prompts rush to web. `http://www.abc.net.au/science/articles/2006/09/06/1734081.htm`, September 06, 2006. Reuters. 6.5.1

[Ter04] Daniel Terdiman. Solution for slashdot effect? `http://www.wired.com/science/discoveries/news/2004/10/65165`, October 1, 2004. 2:00 AM. 6.5.1

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996. 1.3

[Tur03] Vernon Turner. HP utility data center: Enabling enhanced datacenter agility. `http://h30046.www3.hp.com/uploads/whitepapers/IDCwhitepaperHPUDC1.pdf`, June 2003. 2.3.2

[UC01] University of California, Irvine and Carnegie Mellon University. xArch. `http://www.isr.uci.edu/architecture/xarch/`, 2001. 2.1

[Uni96] United States Department of Commerce Institute for Telecommunication Sciences. Telecommunications: Glossary of telecommunication terms. `http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm`, August 7, 1996. Federal Standard 1037C. 3.3.1

[VK02] Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In Garlan et al. [GKW02], pages 73–78. 2.3.1

[VKK01] 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. 2.3.1, 3.3.2

[VS95] Manuela Veloso and Peter Stone. Flecs: Planning with a flexible commitment strategy. *Journal of AI Research (JAIR)*, 3, March 1995. 2.2

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, May 1989. 2.2

[WHC+01] 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 [CDS01]. 2.3.3

[Whi06] Walter White. Personal communication, July 2006. Sys-admin interview. 6.6

[Wik08a] Wikipedia. Control theory — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Control_theory&oldid=202373296`, March 31, 2008. [Online; accessed 2-April-2008]. 2.2

[Wik08b] Wikipedia. Decision theory — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Decision_theory&oldid=201515665`, March 28, 2008. [Online; accessed 2-April-2008]. 2.2

[Wik08c] Wikipedia. Dynamical system — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Dynamical_system&oldid=198931139`, March 17, 2008. [Online; accessed 2-April-2008]. 2

[Wik08d] Wikipedia. Slashdot effect — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Slashdot_effect&oldid=202237428`, March 31, 2008. [Online; accessed 31-January-2008]. 6.5.1, 6.5.4

[Wik08e] Wikipedia. Utility — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Utility&oldid=200699805`, March 17, 2008. [Online; accessed 25-March-2008] See "Expected utility" section. 2.2, 4.3.2

[Wik08f] Wikiversity. System administration. `http://en.wikiversity.org/wiki/Topic:System_Administration`, March 31, 2008. [Online; accessed 14-March-2008] Though this article covers system administration in general, note that the resolution time mentioned are on the order of minutes to hours to days. 8.2

[WLF01] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. *SIGSOFT Software Eng. Notes*, 26(5):21–32, 2001. 2.3.3

[WP01] David Wells and Paul Pazandak. Taming cyber incognito: Surveying dynamic/reconfigurable software landscapes. In CDSA [CDS01]. 2.3.1

[XDP04] Min Xie, Yuan-Shum Dai, and Kim-Leng Poh. *Computing System Reliability:*

Models and Analysis. Springer-Verlag New York, LLC, April 2004. 2.1

[YAM04] YAML.org. The official yaml web site. `http://www.yaml.org/`, December 28, 2004. Announcement of YAML 1.1 Working Draft. 4.3.2

[YGS+04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *Proc. of the 26th International Conf. on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004. 2.3.1, 5.1.3

[You07] Russell J. Yount. Personal communication, April 2, 2007. Discussed the overhauled CMU network architecture. 6.6, 6.7

Appendix A

Rainbow Framework Architectural Style

The Component-and-Connector architecture of Rainbow, described using Acme and modeled in AcmeStudio, is diagrammed in Figure 5.1 with its architectural element types shown below, defined in 3 hierarchical Acme families. BaseRainbowFam defines the basic element types that can be instantiated or extended at any level of representation in the architecture—e.g., generic data, provide and require port types. RainbowFam defines the top-level Rainbow architectural element types instantiated in the above diagram. Element types described in the first-level representations are defined in RainbowFamImpl1a. Types in each family below are listed in the order: component, connector, port (component interface) , and role (connector interface).

Type Name	**Functional Description**
BaseRainbowFam	
DataT	Stores generic data
ModelT	Represents the model of the target system
CallReturnConnT	Generic call-return
ControlConnT (*CallReturnCn*)	Specific call-return type that controls interactions
QueryConnT (*CallReturnCn*)	Specific call-return type that queries data
NotifyConnT	Publish-subscribe, event-based
PollConnT	Point-to-point, event-based push interaction
ProvidePortT	Provides service or data to another component
PublishPortT (*ProvideP*)	Specific event-based type that publishes events
RequirePortT	Requires service or data of another component
SubscribePortT (*RequireP*)	Specific event-based type that subscribes to events
OriginatorRoleT	Role played by providing port of a component
PublisherRoleT (*OriginatorR*)	Specific event-based role played by publishing port
RecipientRoleT	Role type played by the requiring port of a component
SubscriberRoleT (*RecipientR*)	Specific event-based role played by subscribing port
RainbowFam	
AdaptationEngineT	Performs adaptation
ArchAnalyzerT	Evaluates architectural constraints on the model

CustomizerT	Customizes Rainbow components
EffectorT	Propagates and enacts changes in the target system
ExecutorT	Carries out adaptation actions on system via translator
GaugeT	Updates model properties
ModelManagerT	Manages model instance(s), provides model info query
ProbeT	Extracts monitoring information from target system
TargetT	Represents or simulates the target system
TranslatorT	Maintains correspondence between model and system
EffectingConnT (*ControlCn*)	For Effector to effect changes on target system
ChangeNotifyConnT (*NotifyCn*)	Change notification bus, listen for system change requests
GaugeNotifyConnT (*NotifyCn*)	A gauge bus
ProbeNotifyConnT (*NotifyCn*)	A probe bus
RetargetConnT (*NotifyCn*)	Customization notification bus for Customizer
TriggeredAnalyzeConnT (*PollCn*)	Specific polling type, model update triggers analysis
ProbingConnT (*PollCn*)	Specific polling type, instruments the target system
EffectingProvPortT (*ProvideP*)	Enables effecting changes on target system
MonitoringProvPortT (*ProvideP*)	Enables target system states to be monitored
RetargetProvPortT (*ProvideP*)	Customization point to tailor component to specific style
EffectingReqPortT (*RequireP*)	On the Effector, effects system changes
MonitoringReqPortT (*RequireP*)	On the Probe, monitors (extracts) system states
RetargetReqPortT (*RequireP*)	For Customizer to tailor component to specific style
UserPrefReqPortT (*RequireP*)	For Rainbow to receive user preference info
RetargeteeRoleT (*SubscriberR*)	Event bus subscriber role played by retargetee component
RainbowFam-Impl1a	
AcmeModelT (*Model*)	C&C architecture model of target system
EnvironmentModelT (*Model*)	Model about environment in which system executes
QoSPreferenceT (*Data*)	Stores user QoS prefs as utility profiles & preferences
QoSConstraintsT (*QoSPreference*)	Represents rules defined in the architecture model
StitchRepertoireT (*Data*)	Stores strategies, tactics, and tactic meta-information
UtilityDataT (*Data*)	Stores utility curves & weights for strategy selection
ActiveProcessT	Acts as an active process
ChangeSynchronizerT (*ActiveProcess*)	Checks if arch changes in sync with observed system
ConstraintEvaluatorT (*ActiveProcess*)	Evaluates constraints satisfaction of a model
ModelUpdaterT (*ActiveProcess*)	Updates model elements/properties based on gauges
QueryProvisionerT (*ActiveProcess*)	Provides architecture and environment query service
StitchEngineT (*ActiveProcess*)	Responds to trigger, selects strategy, executes tactic
UtilityEvaluatorT (*ActiveProcess*)	Performs utility calculation to select strategy
DataAccessConnT (*CallReturnCn*)	Accesses data from a DataT component

Appendix B

Stitch Grammar

The Stitch language for self-adaptation is presented in Chapter 4. Here we list the full grammar of Stitch, but we have elided nuanced details for working around parsing ambiguities.

```
# Note that the grammar, written in ANTLR, uses the following BNF conventions:
#  - lowercase term identifies a non-terminal
#  - uppercase term identifies a terminal, whose lexeme should be deducible from label, except:
#    > IDENTIFIER ::= (UNDERSCORE)* LETTER (UNDERSCORE | DOT | LETTER | DIGIT | MINUS)*
#    > INTEGER_LIT ::= (DIGIT)+          > FLOAT_LIT ::= (DIGIT)+ (DOT (DIGIT)+)?
#    > STRING_LIT ::= DQUOTE (~'"')* DQUOTE  // non-double-quote characters
#    > CHAR_LIT ::= SQUOTE ~'\'' SQUOTE       // a non-single-quote character

               script ::= MODULE IDENTIFIER SEMICOLON
                          (import)*
                          (function)*
                          (tactic)*
                          (strategy)*
                          EOF ;

               import ::= IMPORT (LIB|MODEL|OP) STRING_LIT importRenameClause? SEMICOLON ;
   importRenameClause ::= LBRACE importRenamePhrase (COMMA importRenamePhrase)* RBRACE ;
   importRenamePhrase ::= IDENTIFIER AS IDENTIFIER ;

             function ::= DEFINE dataType IDENTIFIER ASSIGN expression SEMICOLON ;
             operator ::= identifierPrimary ;

               tactic ::= TACTIC signature LBRACE
                            ( declaration SEMICOLON )*
                            CONDITION LBRACE (booleanExpression SEMICOLON)* RBRACE
                            ACTION LBRACE (statement)* RBRACE
                            EFFECT LBRACE (booleanExpression SEMICOLON)* RBRACE
                          RBRACE ;

             strategy ::= STRATEGY IDENTIFIER
                          LBRACKET booleanExpression RBRACKET
                          LBRACE (function)* (strategyExpr)* RBRACE ;
         strategyExpr ::= IDENTIFIER COLON strategyCond IMPLIES strategyOutcome ;
         strategyCond ::= LPAREN (HASH LBRACKET strategyProbValue RBRACKET)?
                          (booleanExpression | SUCCESS | DEFAULT) RPAREN ;
      strategyOutcome ::= strategyClosedOutcome SEMICOLON
                        | strategyOpenOutcome (AT LBRACKET expression RBRACKET)?
                          LBRACE (strategyExpr)+ RBRACE ;
strategyClosedOutcome ::= DONE | FAIL
                        | DO LBRACKET (IDENTIFIER | INTEGER_LIT)? RBRACKET IDENTIFIER ;
  strategyOpenOutcome ::= IDENTIFIER LPAREN (argExpressionList)? RPAREN
                        | NULLTACTIC ;
    strategyProbValue ::= FLOAT_LIT | IDENTIFIER (LBRACE IDENTIFIER RBRACE)?
```

```
44
45              signature ::= IDENTIFIER
46                            LPAREN (dataType IDENTIFIER (COMMA dataType IDENTIFIER)*)? RPAREN ;
47            declaration ::= dataType IDENTIFIER (ASSIGN expression)?
48                            (COMMA IDENTIFIER (ASSIGN expression)?)* ;
49               dataType ::= OBJECT | INT | FLOAT | BOOLEAN | CHAR | STRING
50                          | SET (LBRACE dataType RBRACE)?
51                          | SEQUENCE (LT dataType GT)?
52                          | RECORD (LBRACKET
53                            (IDENTIFIER (COMMA IDENTIFIER)* (COLON dataType)? SEMICOLON)*
54                            RBRACKET)?
55                          | ENUM (LBRACE (IDENTIFIER (COMMA IDENTIFIER)*)? RBRACE)?
56                          | IDENTIFIER ;
57
58              statement ::= compoundStatement
59                          | declaration SEMICOLON
60                          | assignmentStatement SEMICOLON
61                          | expression SEMICOLON
62                          | operator
63                          | ifThenStatement
64                          | ifThenElseStatement
65                          | whileStatement
66                          | forStatement
67                          | SEMICOLON ;
68      compoundStatement ::= LBRACE (statement)* RBRACE ;
69     assignmentStatement ::= IDENTIFIER ASSIGN expression ;
70         ifThenStatement ::= IF LPAREN booleanExpression RPAREN statement ;
71     ifThenElseStatement ::= IF LPAREN booleanExpression RPAREN statement ELSE statement ;
72          whileStatement ::= WHILE LPAREN booleanExpression RPAREN statement ;
73            forStatement ::= FOR LPAREN
74                           ( (declaration | assignmentList)? SEMICOLON
75                             (booleanExpression)? SEMICOLON
76                             (assignmentList)?
77                           | parameterDeclaration COLON expression
78                           ) RPAREN statement ;
79
80      argExpressionList ::= expression (COMMA expression)* ;
81         assignmentList ::= assignmentStatement (COMMA assignmentStatement)* ;
82             expression ::= booleanExpression ;
83      booleanExpression ::= quantifiedExpression | impliesExpression ;
84   quantifiedExpression ::= ( FORALL | EXISTS (UNIQUE)? ) quantifierDeclaration
85                            IN (setExpression | identifierPrimary) BAR booleanExpression ;
86  quantifierDeclaration ::= IDENTIFIER (COMMA IDENTIFIER)* COLON dataType ;
87      impliesExpression ::= iffExpression (IMPLIES impliesExpression)? ;
88           iffExpression ::= logicalOrExpression (IFF iffExpression)? ;
89    logicalOrExpression ::= logicalAndExpression (LOGICAL_OR logicalAndExpression)* ;
90   logicalAndExpression ::= equalityExpression (LOGICAL_AND equalityExpression)* ;
91     equalityExpression ::= relationalExpression ((NE | EQ) relationalExpression)* ;
92   relationalExpression ::= additiveExpression ((LT | LE | GE | GT) additiveExpression)* ;
93     additiveExpression ::= multiplicativeExpr ((PLUS | MINUS) multiplicativeExpr)* ;
94     multiplicativeExpr ::= unaryExpression ((STAR | SLASH | MOD) unaryExpression)* ;
95        unaryExpression ::= INCR unaryExpression
96                          | DECR unaryExpression
97                          | UNARY_MINUS unaryExpression
98                          | UNARY_PLUS unaryExpression
99                          | LOGICAL_NOT unaryExpression
100                         | primaryExpression ;
101     primaryExpression ::= identifierPrimary | setExpression | constant
102                         | LPAREN assignmentExpression RPAREN ;
103     identifierPrimary ::= IDENTIFIER (LPAREN (argExpressionList)? RPAREN)? ;
104         setExpression ::= setConstructor | literalSet ;
105        setConstructor ::= LBRACE SELECT quantifierDeclaration
106                           IN (setExpression | identifierPrimary) BAR booleanExpression RBRACE ;
107            literalSet ::= LBRACE ((identifierPrimary | constant)
108                           (COMMA (identifierPrimary | constant))*)? RBRACE ;
109              constant ::= INTEGER_LIT|FLOAT_LIT|STRING_LIT|CHAR_LIT | TRUE|FALSE|NULL ;
```

Appendix C

Znn.com Customization Content

This appendix lists the complete customizations of Rainbow for the Znn.com system. Confer Chapter 5 to see how individual Rainbow components are customized.

Acme Model Core of this architecture-based self-adaptation approach, it customizes the Model Manager and the Architecture Evaluator. The model consists of definitions of one or more styles, defined by the *style writer*, and an instance that extends those styles. Section 5.1.2.

```
1  import TargetEnvType.acme;
2
3  Family ZNewsFam extends EnvType with {
4      Property MIN_RESPTIME : float;
5      Property MAX_RESPTIME : float;
6      Property TOLERABLE_PERCENT_UNHAPPY : float;
7      Property UNHAPPY_GRADIENT_1 : float;
8      Property UNHAPPY_GRADIENT_2 : float;
9      Property UNHAPPY_GRADIENT_3 : float;
10     Property FRACTION_GRADIENT_1 : float;
11     Property FRACTION_GRADIENT_2 : float;
12     Property FRACTION_GRADIENT_3 : float;
13     Property MIN_UTIL : float;
14     Property MAX_UTIL : float;
15     Property MAX_FIDELITY_LEVEL : int;
16     Property THRESHOLD_FIDELITY : int;
17     Property THRESHOLD_COST : float;
18     Component Type ServerT extends ArchElementT with {
19         Property deploymentLocation : string << default : string = "localhost"; >>;
20         Property load : float << default : float = 0.0; >> ;
21         Property reqServiceRate : float << default : float = 0.0; >> ;
22         Property byteServiceRate : float << default : float = 0.0; >> ;
23         Property fidelity : int << HIGH:int = 5; LOW:int = 1; default:int = 5; >>;
24         Property cost : float << default : float = 1.0; >> ;
25         Property lastPageHit : Record [uri : string; cnt : int; kbytes : float; ];
26         rule anotherConstraint = heuristic self.load <= MAX_UTIL;
27     }
28     Component Type ClientT extends ArchElementT with {
29         Property deploymentLocation : string << default : string = "localhost"; >>;
30         Property experRespTime : float << default : float = 100.0; >> ;
31         Property reqRate : float << default : float = 0.0; >> ;
32         rule primaryConstraint = invariant self.experRespTime <= MAX_RESPTIME;
33         rule reverseConstraint = heuristic self.experRespTime >= MIN_RESPTIME;
34     }
35     Component Type ProxyT extends ArchElementT with {
36         Property deploymentLocation : string << default : string = "localhost"; >>;
37         Property load : float << default : float = 0.0; >> ;
```

```
38      }
39      Connector Type ProxyConnT extends ArchConnT with {
40          Role req : RequestorRoleT = new RequestorRoleT extended with { }
41          Role rec : ReceiverRoleT = new ReceiverRoleT extended with { }
42      }
43      Connector Type HttpConnT extends ArchConnT with {
44          Property bandwidth : float << default : float = 0.0; >> ;
45          Property latency : float << default : float = 0.0; >> ;
46          Property numReqsSuccess : int << default : int = 0; >> ;
47          Property numReqsRedirect : int << default : int = 0; >> ;
48          Property numReqsClientError : int << default : int = 0; >> ;
49          Property numReqsServerError : int << default : int = 0; >> ;
50          Property latencyRate : float;
51          Role req : RequestorRoleT = new RequestorRoleT extended with { }
52          Role rec : ReceiverRoleT = new ReceiverRoleT extended with { }
53      }
54      Port Type HttpReqPortT extends ArchPortT with { }
55      Port Type HttpPortT extends ArchPortT with { }
56      Port Type ProxyForwardPortT extends ArchPortT with { }
57      Role Type RequestorRoleT extends ArchRoleT with { }
58      Role Type ReceiverRoleT extends ArchRoleT with { }
59  }
60
61  System ZNewsSys : ZNewsFam = {
62      Property MIN_RESPTIME : float = 100.0;
63      Property MAX_RESPTIME : float = 1000.0;
64      Property UNHAPPY_GRADIENT_1 : float = 0.1;
65      Property UNHAPPY_GRADIENT_2 : float = 0.2;
66      Property UNHAPPY_GRADIENT_3 : float = 0.5;
67      Property FRACTION_GRADIENT_1 : float = 0.2;
68      Property FRACTION_GRADIENT_2 : float = 0.4;
69      Property FRACTION_GRADIENT_3 : float = 1.0;
70      Property TOLERABLE_PERCENT_UNHAPPY : float = 0.4;
71      Property MIN_UTIL : float = 0.1;
72      Property MAX_UTIL : float = 0.75;
73      Property MAX_FIDELITY_LEVEL : int = 5;
74      Property THRESHOLD_FIDELITY : int = 2;
75      Property THRESHOLD_COST : float = 4.0;
76      Component s0 : ServerT = new ServerT extended with {
77          Property deploymentLocation = "oracle";
78          Property cost = 0.9;
79          Property fidelity = 3;
80          Property load = 0.198;
81          Property isArchEnabled = true;
82          Port http0 : HttpPortT = new HttpPortT extended with { }
83      }
84      Component s1 : ServerT = new ServerT extended with { ... }
85      Component s2 : ServerT = new ServerT extended with { ... }
86      Component s3 : ServerT = new ServerT extended with { ... }
87      Component lbproxy : ProxyT = new ProxyT extended with {
88          Property deploymentLocation = "127.0.0.1";
89          Property isArchEnabled = true;
90          Property load = 0.01;
91          Port fwd0 : ProxyForwardPortT = new ProxyForwardPortT extended with { }
92          Port fwd1 : ProxyForwardPortT = new ProxyForwardPortT extended with { }
93          Port fwd2 : ProxyForwardPortT = new ProxyForwardPortT extended with { }
94          Port fwd3 : ProxyForwardPortT = new ProxyForwardPortT extended with { }
95          Port http0 : HttpPortT = new HttpPortT extended with { }
96          Port http1 : HttpPortT = new HttpPortT extended with { }
97          Port http2 : HttpPortT = new HttpPortT extended with { }
98      }
99      Component c0 : ClientT = new ClientT extended with {
100         Property deploymentLocation = "127.0.0.1";
101         Property isArchEnabled = true;
102         Property experRespTime = 2360.2585;
103         Port p0 : HttpReqPortT = new HttpReqPortT extended with { }
```

```
104       }
105       Component c1 : ClientT = new ClientT extended with { ... }
106       Component c2 : ClientT = new ClientT extended with { ... }
107       Connector proxyconn0 : ProxyConnT = new ProxyConnT extended with {
108           Property isArchEnabled = true;
109           Role req  = { Property isArchEnabled = true; }
110           Role rec  = { Property isArchEnabled = true; }
111       }
112       Connector proxyconn1 : ProxyConnT = new ProxyConnT extended with { }
113       Connector proxyconn3 : ProxyConnT = new ProxyConnT extended with { }
114       Connector proxyconn2 : ProxyConnT = new ProxyConnT extended with { }
115       Connector conn : HttpConnT = new HttpConnT extended with {
116           Property latencyRate = 0.0;
117           Property isArchEnabled = true;
118           Role req  = { Property isArchEnabled = true; }
119           Role rec  = { Property isArchEnabled = true; }
120       }
121       Connector conn0 : HttpConnT = new HttpConnT extended with { ... }
122       Connector conn1 : HttpConnT = new HttpConnT extended with { ... }
123       Attachment lbproxy.fwd0 to proxyconn0.req;
124       Attachment s0.http0 to proxyconn0.rec;
125       Attachment lbproxy.http0 to conn0.rec;
126       Attachment c1.p0 to conn.req;
127       Attachment c2.p0 to conn1.req;
128       Attachment lbproxy.http2 to conn1.rec;
129       Attachment lbproxy.http1 to conn.rec;
130       Attachment c0.p0 to conn0.req;
131       Attachment s2.http0 to proxyconn2.rec;
132       Attachment lbproxy.fwd1 to proxyconn1.req;
133       Attachment s1.http0 to proxyconn1.rec;
134       Attachment s3.http0 to proxyconn3.rec;
135       Attachment lbproxy.fwd3 to proxyconn3.req;
136       Attachment lbproxy.fwd2 to proxyconn2.req;
137   }
```

Probe Specification The *system adapter* defines these available probes for the target system using Yaml syntax. Section 5.1.3.

```
1   vars:
2     _probes.commonPath: "${rainbow.path}/system/probes"
3   probes:
4     ClientProxyProbe0:
5       alias: clientproxy
6       location: "${customize.system.target.0}"
7       type: java
8       javaInfo:
9         class: org.sa.rainbow.translator.znews.probes.ClientProxyProbe
10        period: 2000
11        args.length: 1
12        args.0: "http://delegate.oracle/"
13    PingRTTProbe1:
14      alias: pingrtt
15      location: "${customize.system.target.1}"
16      type: java
17      javaInfo:
18        class: org.sa.rainbow.translator.znews.probes.PingRTTProbe
19        period: 1500
20        args.length: 1
21        args.0: "${rainbow.master.location.host}"
22        args.1: "${customize.system.target.2}"
23    LoadProbe1:
24      alias: load
25      location: "${customize.system.target.1}"
26      type: script
```

```
27      scriptInfo:
28        path     : "${_probes.commonPath}/loadProbe.pl"
29        argument: "-k -s"
30    FidelityProbe1:
31      alias: fidelity
32      location: "${customize.system.target.1}"
33      type: script
34      scriptInfo:
35        path     : "${_probes.commonPath}/fidelityProbe.pl"
36        argument: "-k -s"
37    ApacheTopProbe1:
38      alias: apachetop
39      location: "${customize.system.target.1}"
40      type: script
41      scriptInfo:
42        path     : "${_probes.commonPath}/apachetopProbe.pl"
43        argument: "-k -s"
44    DiskIOProbe1:
45      alias: diskio
46      location: "${customize.system.target.1}"
47      type: script
48      scriptInfo:
49        path     : "${_probes.commonPath}/diskIOProbe.pl"
50        argument: "-k -s"
```

Gauge Specification The *gauge writer* defines these available gauges for the Translation Infrastructure using Yaml. Section 5.1.3.

```
1   # Gauge Type and Gauge Instance Specifications
2   #  - time periods generally in milliseconds
3   gauge-types:
4     ResponseTimeGaugeT:
5       values:
6         end2endRespTime : double
7       setupParams:
8         targetIP:
9           type:    String
10          default: "localhost"
11        beaconPeriod:
12          type:    long
13          default: 30000
14        javaClass:
15          type:    String
16          default: "org.sa.rainbow.translator.znews.gauges.End2EndRespTimeGauge"
17      configParams:
18        samplingFrequency:
19          type:    long
20          default: 1000
21        targetProbeType:
22          type:    String
23          default: ~
24      comment: "ResponseTimeGaugeT reports end-to-end response time from client proxy
            ."
25    ApacheTopGaugeT:
26      values:
27        reqServiceRate      : double
28        byteServiceRate     : double
29        numReqsSuccess      : int
30        numReqsRedirect     : int
31        numReqsClientError  : int
32        numReqsServerError  : int
33        pageHit : String
34      setupParams:
35        targetIP:
```

```
36          type:     String
37          default: "localhost"
38        beaconPeriod:
39          type:     long
40          default: 20000
41        javaClass:
42          type:     String
43          default: "org.sa.rainbow.translator.gauges.ApacheTopGauge"
44      configParams:
45        samplingFrequency:
46          type:     long
47          default: 1000
48        targetProbeType:
49          type:     String
50          default: ~
51      comment: "ApacheTopGaugeT reports Apache server properties via 'apachetop'."
52    DiskIOGaugeT:
53      values:
54        transferRate : double
55        readRate     : double
56        writeRate    : double
57      setupParams:
58        targetIP:
59          type:     String
60          default: "localhost"
61        beaconPeriod:
62          type:     long
63          default: 20000
64        javaClass:
65          type:     String
66          default: "org.sa.rainbow.translator.gauges.DiskIOGauge"
67      configParams:
68        samplingFrequency:
69          type:     long
70          default: 1000
71        targetProbeType:
72          type:     String
73          default: ~
74      comment: "DiskIOGaugeT reports disk I/O stat of host (read/write in KBps)."
75    LoadGaugeT:
76      values:
77        load : double
78      setupParams:
79        targetIP:
80          type:     String
81          default: "localhost"
82        beaconPeriod:
83          type:     long
84          default: 20000
85        javaClass:
86          type:     String
87          default: "org.sa.rainbow.translator.gauges.CpuLoadGauge"
88      configParams:
89        samplingFrequency:
90          type:     long
91          default: 1000
92        targetProbeType:
93          type:     String
94          default: ~
95      comment: "LoadGaugeT reports CPU load of target host"
96    FidelityGaugeT:
97      values:
98        fidelity : int
99      setupParams:
100       targetIP:
101         type:     String
```

```
102           default: "localhost"
103         beaconPeriod:
104           type:    long
105           default: 30000
106         javaClass:
107           type:    String
108           default: "org.sa.rainbow.translator.gauges.FidelityGauge"
109       configParams:
110         samplingFrequency:
111           type:    long
112           default: 2500
113         targetProbeType:
114           type:    String
115           default: ~
116       comment: "FidelityGaugeT reports fidelity level of served content"
117     LatencyGaugeT:
118       values:
119         latency : double
120       setupParams:
121         targetIP:
122           type:    String
123           default: "localhost"
124         beaconPeriod:
125           type:    long
126           default: 20000
127         javaClass:
128           type:    String
129           default: "org.sa.rainbow.translator.znews.gauges.RtLatencyMultiHostGauge"
130       configParams:
131         samplingFrequency:
132           type:    long
133           default: 1500
134         targetProbeType:
135           type:    String
136           default: ~
137       comment: "LatencyGaugeT reports latency on a connection"
138     LatencyRateGaugeT:
139       values:
140         latencyRate : double
141       setupParams:
142         targetIP:
143           type:    String
144           default: "localhost"
145         beaconPeriod:
146           type:    long
147           default: 20000
148         javaClass:
149           type:    String
150           default: "o.s.rainbow.translator.znews.gauges.RtLatencyRateMultiHostGauge"
151       configParams:
152         samplingFrequency:
153           type:    long
154           default: 1500
155         targetProbeType:
156           type:    String
157           default: ~
158       comment: "LatencyRateGaugeT reports latency rate of change on a connection"
159   gauge-instances:
160     EERTG1:
161       type:   ResponseTimeGaugeT
162       model: "ZNewsSys:Acme"
163       mappings:
164         "end2endRespTime(delegate.oracle)" : c0.experRespTime
165       setupValues:
166         targetIP: "${customize.system.target.0}"
167       configValues:
```

```
168         samplingFrequency: 1000
169         targetProbeType   : clientproxy
170       comment: "EERTG1 is associated with client component of ZNewsSys in Acme."
171     ATG1:
172       type:   ApacheTopGaugeT
173       model: "ZNewsSys:Acme"
174       mappings:
175         reqServiceRate      : s0.reqServiceRate
176         byteServiceRate     : s0.byteServiceRate
177         numReqsSuccess      : conn0.numReqsSuccess
178         numReqsRedirect     : conn0.numReqsRedirect
179         numReqsClientError  : conn0.numReqsClientError
180         numReqsServerError  : conn0.numReqsServerError
181         pageHit : s0.lastPageHit
182       setupValues:
183         targetIP: "${customize.system.target.1}"
184       configValues:
185         samplingFrequency: ~
186         # Leave details unspecified (null) to use type-level default value
187         targetProbeType   : apachetop
188       comment: "ATG1 is associated with comp s0 and conn conn0 of ZNewsSys in Acme."
189     DioG1:
190       type:   DiskIOGaugeT
191       model: "ZNewsSys:Acme"
192       mappings:
193         transferRate : s0.diskXferRate
194         readRate : s0.diskReadRate
195         writeRate : s0.diskWriteRate
196       setupValues:
197         targetIP: "${customize.system.target.1}"
198       configValues:
199         samplingFrequency: 1000
200         targetProbeType   : diskio
201       comment: "DioG1 is associated with component s0 of ZNewsSys in Acme."
202     LoG0:
203       type:   LoadGaugeT
204       model: "ZNewsSys:Acme"
205       mappings:
206         load : s0.load
207       setupValues:
208         targetIP: "${customize.system.target.0}"
209       configValues:
210         samplingFrequency: 1000
211         targetProbeType   : load
212       comment: "LoG0 is associated with component s0 of ZNewsSys in Acme."
213     LoG1: ...
214     LoG2: ...
215     LoG3: ...
216     FiG1:
217       type:   FidelityGaugeT
218       model: "ZNewsSys:Acme"
219       mappings:
220         fidelity : s0.fidelity
221       setupValues:
222         targetIP: "${customize.system.target.1}"
223       configValues:
224         samplingFrequency: 2500
225         targetProbeType   : fidelity
226       comment: "FiG1 is associated with component s0 of ZNewsSys in Acme."
227     LatG1:
228       type:   LatencyGaugeT
229       model: "ZNewsSys:Acme"
230       mappings:
231         "latency(${rainbow.master.location.host})" : conn0.latency
232         "latency(@{ZNewsSys.s1.deploymentLocation})" : conn1.latency
233       setupValues:
```

```
234         targetIP: "${customize.system.target.1}"
235       configValues:
236         samplingFrequency: 1500
237         targetProbeType   : pingrtt
238       comment: "LatG1 is associated with connectors of ZNewsSys in Acme."
239     LatRoCG1:
240       type:  LatencyRateGaugeT
241       model: "ZNewsSys:Acme"
242       mappings:
243         "latencyRate(${rainbow.master.location.host})" : conn0.latencyRate
244         "latencyRate(@{ZNewsSys.s1.deploymentLocation})" : conn1.latencyRate
245       setupValues:
246         targetIP: "${customize.system.target.1}"
247       configValues:
248         samplingFrequency: 1500
249         targetProbeType   : pingrtt
250       comment: "LatRoCG1 is associated with connectors of ZNewsSys in Acme."
```

Effector Specification The *system adapter* defines these available effectors in the target system using Yaml. Section 5.1.3.

```
1  vars:
2    _effectors.commonPath: "${rainbow.path}/system/effectors"
3  effectors:
4  # test from GUI with <host>, SetFidelity, fidelity=<1|3|5>
5    setFidelity1:
6      location: "${customize.system.target.1}"
7      type: script
8      scriptInfo:
9        path     : "${_effectors.commonPath}/changeFidelity.pl"
10       argument: "-l {fidelity}"
11   activateServer1:
12     location: "${customize.system.target.1}"
13     type: script
14     scriptInfo:
15       path     : "${_effectors.commonPath}/turnServer.pl"
16       argument: "-s on"
17   deactivateServer1:
18     location: "${customize.system.target.1}"
19     type: script
20     scriptInfo:
21       path     : "${_effectors.commonPath}/turnServer.pl"
22       argument: "-s off"
23   randomReject1:
24     location: "${customize.system.target.1}"
25     type: script
26     scriptInfo:
27       path     : "${_effectors.commonPath}/setRandomReject.pl"
28       argument: "-r {frequency}"
29   killDelegate1:
30     location: "${customize.system.target.1}"
31     type: java
32     javaInfo:
33       class    : org.sa.rainbow.translator.effectors.KillDelegateEffector
```

Stitch Scripts The *tactic writer* defines tactics and *strategy writer* composes strategies in the Stitch syntax to produce adaptation scripts, used by the Adaptation Manager and executed by the Strategy Executor to remedy the target system. Section 5.1.5.

```
1  module newssite.tactics;
2  import model "ZNewsSys.acme" { ZNewsSys as M, ZNewsFam as T };
```

```
3  import op "znews1.operator.ArchOp" { ArchOp as S };
4  import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
5
6  // Enlist n free servers into service pool.
7  // Utility: [v] R; [^] C; [<>] F
8  tactic enlistServers (int n) {
9    condition {  // some client should be experiencing high response time
10     exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
11     // there should be enough available server resources
12     Model.availableServices(T.ServerT) >= n;
13   }
14   action {
15     set servers = Set.randomSubset(Model.findServices(T.ServerT), n);
16     for (T.ServerT freeSvr : servers) {
17       S.activateServer(freeSvr);
18     }
19   }
20   effect {  // response time should decrease below threshold
21     forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
22   }
23 }
24
25 // Deactivate n servers from service pool into free pool.
26 // Utility: [^] R; [v] C; [<>] F
27 tactic dischargeServers (int n) {
28   condition {  // there should be NO client with high response time
29     forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
30     // there should be enough servers to discharge
31     Set.size({ select s : T.ServerT in M.components | s.load < M.MIN_UTIL }) >= n;
32   }
33   action {
34     set lowUtilSvrs = {select s : T.ServerT in M.components | s.load < M.MIN_UTIL};
35     set subLowUtilSvrs = Set.randomSubset(lowUtilSvrs, n);
36     for (T.ServerT s : subLowUtilSvrs) {
37       S.deactivateServer(s);
38     }
39   }
40   effect {  // still NO client with high response time
41     forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
42   }
43 }
44
45 // Lowers fidelity by integral steps for percent of requests.
46 // Utility: [v] R; [v] C; [v] F
47 tactic lowerFidelity (int step, float fracReq) {
48   condition {  // some client should be experiencing high response time
49     exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
50     // exists server with fidelity to lower
51     exists s : T.ServerT in M.components | s.fidelity > step;
52   }
53   action {
54     // retrieve set of servers who still have enough fidelity grade to lower
55     set servers = { select s : T.ServerT in M.components | s.fidelity > step };
56     for (T.ServerT s : servers) {
57         S.setFidelity(s, s.fidelity - step);
58     }
59   }
60   effect {  // response time decreasing below threshold should result
61     forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
62   }
63 }
64
65 // Raises fidelity by integral steps for percent of requests.
66 // Utility: [^] R; [^] C; [^] F
67 tactic raiseFidelity (int step, float fracReq) {
68   condition {  // there should be NO client with high response time
```

```
69      forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
70      // there exists some client with below low-threshold response time
71      exists c : T.ClientT in M.components | c.experRespTime < M.MIN_RESPTIME;
72    }
73    action {
74      // first find the lowest fidelity set
75      set servers = { select s : T.ServerT in M.components | s.fidelity <= M.
            MAX_FIDELITY_LEVEL - step};
76      int lowestFidelity = M.MAX_FIDELITY_LEVEL;
77      for (T.ServerT s : servers) {
78        if (s.fidelity < lowestFidelity) {
79          lowestFidelity = s.fidelity;
80        }
81      }
82      // find only servers with this lowest fidelity setting, and raise fidelity
83      servers = {select s:T.ServerT in M.components | s.fidelity <= lowestFidelity};
84      for (T.ServerT s : servers) {
85        S.setFidelity(s, java.lang.Math.min(s.fidelity+step, M.MAX_FIDELITY_LEVEL));
86      }
87    }
88    effect {  // still NO client with high response time
89      forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
90    }
91  }
```

```
1   module newssite.strategies;
2   import lib "newssiteTactics.s";
3
4   define boolean styleApplies =
5       Model.hasType(M, "ClientT") && Model.hasType(M, "ServerT");
6   define boolean cViolation =
7       exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
8
9   define set servers = {select s : T.ServerT in M.components | true};
10  define set unhappyClients =
11      {select c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME};
12  define int numClients = Set.size({select c : T.ClientT in M.components | true});
13  define int numUnhappy = Set.size(unhappyClients);
14  define float numUnhappyF = 1.0*numUnhappy;
15
16  define boolean hiLoad = exists s : T.ServerT in M.components | s.load > M.MAX_UTIL;
17  define boolean hiRespTime =
18      exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
19  define boolean lowRespTime =
20      exists c : T.ClientT in M.components | c.experRespTime < M.MIN_RESPTIME;
21  define float totalCost = Model.sumOverProperty("cost", servers);
22  define boolean hiCost = totalCost >= M.THRESHOLD_COST;
23  define float avgFidelity =
24      Model.sumOverProperty("fidelity", servers) / Set.size(servers);
25  define boolean lowFi = avgFidelity < M.THRESHOLD_FIDELITY;
26
27  // While it encounters high experienced response time, this simple strategy
28  // first enlists one new server, then lowers fidelity one step, then quits
29  strategy SimpleReduceResponseTime
30  [ styleApplies && cViolation ] {
31    t0: (/*hiLoad*/ cViolation) -> enlistServers(1) @[1000 /*ms*/] {
32      t1: (!cViolation) -> done;
33      t2: (/*hiRespTime*/ cViolation) -> lowerFidelity(2, 100) @[3000 /*ms*/] {
34        t2a: (!cViolation) -> done;
35        t2b: (default) -> TNULL;  // in this case, we have no more steps to take
36      }
37    }
38  }
39
40  // This strategy looks for a percentage of clients with anomalous experienced
41  // response time, in which case it enlists a few servers in sequence, then
```

```
42  // lowers fidelity a few steps, then starts delaying Clients
43  strategy SmarterReduceResponseTime
44  [ styleApplies && cViolation ] {
45    define boolean unhappy= numUnhappyF/numClients > M.TOLERABLE_PERCENT_UNHAPPY;
46
47    t0: (unhappy) -> enlistServers(1) @[500 /*ms*/] {
48      t1: (!cViolation) -> done;
49      t2: (unhappy) -> enlistServers(1) @[2000 /*ms*/] {
50        t2a: (!cViolation) -> done;
51        t2b: (unhappy) -> lowerFidelity(2, 100) @[2000 /*ms*/] {
52          t2b1: (!cViolation) -> done;
53          t2b2: (unhappy) -> do[1] t2;
54          t2b3: (default) -> TNULL;  // in this case, we have no more steps to take
55        }
56      }
57    }
58  }
59
60  // This experimental strategy has the sophistication of reducing fidelity
61  // for a percentage of requests depending on percentage of unhappy clients
62  strategy SophisticatedReduceResponseTime
63  [ styleApplies && cViolation && M.SUPPORT_FRACTION_GRADIENT ] {
64    define boolean unhappy1 = numUnhappyF/numClients > M.UNHAPPY_GRADIENT_1; // 10%
65    define boolean unhappy2 = numUnhappyF/numClients > M.UNHAPPY_GRADIENT_2; // 25%
66    define boolean unhappy3 = numUnhappyF/numClients > M.UNHAPPY_GRADIENT_3; // 50%
67
68    t1: (hiLoad) -> enlistServers(2) @[500 /*ms*/] {
69      t1a: (!cViolation) -> done;
70      t1b: (default) -> TNULL;
71    }
72    t2: (unhappy1) -> lowerFidelity(1, M.FRACTION_GRADIENT_1) @[500 /*ms*/] {
73      t2a: (!cViolation) -> done;
74      t2b: (default) -> do[1] t1;
75    }
76    t3: (unhappy2) -> lowerFidelity(2, M.FRACTION_GRADIENT_2) @[500 /*ms*/] {
77      t3a: (!cViolation) -> done;
78      t3b: (default) -> do[1] t1;
79    }
80    t4: (unhappy3) -> lowerFidelity(4, M.FRACTION_GRADIENT_3) @[500 /*ms*/] {
81      t4a: (!cViolation) -> done;
82      t4b: (unhappy3) -> enlistServers(2) @[1000 /*ms*/] {
83        t4b1: (!cViolation) -> done;
84        t4b2: (default) -> do[1] t1;
85      }
86      t4c: (default) -> do[1] t1;
87    }
88  }
89
90  // Triggered by total server costs above threshold, it reduces # active servers
91  strategy ReduceOverallCost
92  [ styleApplies && hiCost ] {
93    t0: (hiCost) -> dischargeServers(1) @[2000 /*ms*/] {
94      t1: (!hiCost) -> done;
95      t2: (lowRespTime && hiCost) -> do[2] t0;
96      t3: (default) -> TNULL;
97    }
98  }
99
100 // Triggered by overall fidelity below threshold; it raises server fidelity
101 strategy ImproveOverallFidelity
102 [ styleApplies && lowFi ] {
103   t0: (lowFi) -> raiseFidelity(2, 100) @[800 /*ms*/] {
104     t1: (!lowFi) -> done;
105     t2: (lowRespTime && lowFi) -> do[1] t0;
106     t3: (default) -> TNULL;
107   }
```

```
108 }
```

Utility Specification The *adaptation integrator* defines, in Yaml, from business input, the quality dimension utility profiles and business preferences over them. Section 5.1.5.

```
utilities:
  uR:
    label: "Average Response Time"
    mapping: "[EAvg]ClientT.experRespTime"
    description: "R, client experienced response time (ms), float arch property"
    utility:
      0: 1.00
      100: 1.00
      200: 0.99
      500: 0.90
      1000: 0.75
      1500: 0.50
      2000: 0.25
      4000: 0.00
  uF:
    label: "Average Fidelity"
    mapping: "[EAvg]ServerT.fidelity"
    description: "F, server content fidelity level, int arch property"
    utility:
      1: 0.80
      5: 1.00
  uC:
    label: "Average Server Cost"
    mapping: "[EAvg]ServerT.cost"
    description: "C, server cost in unit/hr, average of float arch property"
    utility:
      0: 1.00
      1: 0.90
      5: 0.20
      10: 0.00
  uD:
    label: "Service Disruption"
    mapping: "[EAvg]ServerT.rejectedRequests"
    description: "D, service disruption measured by requests rejected"
    utility:
      1: 1.00
      2: 0.98
      3: 0.00
      5: 0.00
  uSF:
    label: "Historical Strategy Failure"
    mapping: "[EAvg]Strategy.rateFailure"
    description: "Rate of failure of a strategy; must enable trackStrategy"
    utility:
      0: 1.00
      0.5: 0.01
      1: 0.00

# Weighted utility preferences, each set should sum to 1.  3 scenarios:
#   1.) Normal request rate -> Sustained, peak request rate -> Normal rate
#   2.) Normal request rate -> Transient, peak request rate -> Normal rate
#   3.) Any request rate -> Abnormally high rate (e.g., Ramsey case) -> Any rate
weights:
  scenario 1:
    uR: 0.3
    uF: 0.4
    uC: 0.2
    uSF: 0.1
  scenario 2:
```

```
60      uR: 0.5
61      uF: 0.25
62      uC: 0.15
63      uSF: 0.1
64    scenario 3:
65      uR: 0.45
66      uF: 0.3
67      uC: 0.1
68      uSF: 0.15
69
70  # Tactic quality attribute vectors
71  vectors:
72  # Utility: [v] R; [^] C; [<>] F
73  # assume each server will drop response time by 1000 ms and increase cost by 1 unit
74    enlistServers:
75      uR: -1000
76      uF: 0
77      uC: +1.00
78  # Utility: [^] R; [v] C; [<>] F
79    dischargeServers:
80      uR: +1000
81      uF: 0
82      uC: -1.00
83  # Utility: [v] R; [v] C; [v] F
84  # assume each level of fidelity reduces response time by 500 ms, cost by 10%
85    lowerFidelity:
86      uR: -500
87      uF: -2
88      uC: -0.10
89  # Utility: [^] R; [^] C; [^] F
90    raiseFidelity:
91      uR: +500
92      uF: +2
93      uC: +0.10
```

Mapping Specification The *adaptation integrator* defines this mapping of architectural style operators to target system effectors, used by the Translator to fulfill Strategy Executor change requests. Section 5.1.6.

```
1  setFidelity: changeState
2  activateServer: start
3  deactivateServer: stop
```

Rainbow Oracle Properties To initialize the Rainbow Oracle and Delegates, the *adaptation integrator* defines these configuration parameters, loaded with one-time variable substitutions by singleton class, org.sa.rainbow.Rainbow, which provides access to common Rainbow properties. Section 5.1.1.

```
1  ###############################################################################
2  # Target:   ZNews case study system with Probes, Gauges, and Effectors implemented
3  #           (rainbow.target = znews1-d)
4  # Framework-defined special properties:
5  #     rainbow.path - the canonical path to the target configuration location
6  ###############################################################################
7  ###
8  # Default values for location-specific properties taken from this file if
9  # rainbow-<host>.properties file does not specify a value.
10
11 ### Utility mechanism configuration
12 #- Config for Log4J, with levels:  OFF,FATAL,ERROR,WARN,INFO,DEBUG,TRACE,ALL
```

```
13 logging.level = INFO
14 event.log.path = log
15 logging.path = ${event.log.path}/rainbow.out
16 monitoring.log.path = ${event.log.path}/rainbow-data.log
17 # (default)
18 #logging.pattern = "%d{ISO8601/yyyy-MM-dd HH:mm:ss.SSS} [%t] %p %c %x - %m%n"
19 #logging.max.size = 1024
20 #logging.max.backups = 5
21
22 ### Rainbow component customization
23 ## Rainbow host info and communication infrastructure
24 #- Location information of the master and this deployment
25 rainbow.master.location.host = localhost
26 #- Location information of the deployed delegate
27 rainbow.deployment.location = ${rainbow.master.location.host}
28 #- Rainbow service port
29 rainbow.service.port = 9210
30 #- default registry port; change if port-tunneling
31 rainbow.master.location.port = 1099
32 #- OS platform, supported modes are:  cygwin | linux
33 #  Use "cygwin" for Windows, "linux" for MacOSX
34 rainbow.deployment.environment = linux
35 #- Event infrastructure, type of event middleware: rmi | jms | que
36 rainbow.event.service = rmi
37 #- JMS/JBoss-specific configurations
38 #event.context.factory = org.jnp.interfaces.NamingContextFactory
39 #event.provider.url = ${rainbow.master.location.host}:1099
40 #event.url.prefixes = org.jboss.naming:org.jnp.interfaces
41 ## RainbowDelegate and ProbeBusRelay configurations
42 rainbow.delegate.id = RainbowDelegate@${rainbow.deployment.location}
43 rainbow.delegate.beaconperiod = 5000
44 rainbow.delegate.startProbesOnInit = false
45 probebus.relay.id = ProbeBusRelay@${rainbow.deployment.location}
46 #- uncomment to enable file-based communication with the ProbeBus Relay
47 #probebus.relay.file = ${event.log.path}/relay.log
48
49 ## Model Manager customization
50 #- Arch model file
51 customize.model.path = model/ZNewsSys.acme
52 #- Alpha factor for exponential average, expAvg = (1-alpha)*expAvg + alpha*newVal
53 customize.model.expavg.alpha = 0.30
54 customize.model.evaluate.period = 2000
55 ## Translator customization
56 #- Gauge spec
57 customize.gauges.path = model/gauges.yml
58 #- Probe spec
59 customize.probes.path = system/probes.yml
60 #- Operator spec as mapping to effector
61 customize.archop.map.path = model/op.map
62 #- Effector spec
63 customize.effectors.path = system/effectors.yml
64 ## Adaptation Manager
65 #- Directory of Stitch adaptation script
66 customize.scripts.path = stitch
67 #- Utilities description file and Strategy evaluation configuration
68 customize.utility.path = stitch/utilities.yml
69 customize.utility.trackStrategy = uSF
70 customize.utility.scenario = scenario 2
71 ## System configuration information
72 customize.system.target.0 = ${rainbow.master.location.host}
73 customize.system.target.1 = oracle
74 customize.system.target.size = 2
```

Appendix D

Additional Thesis Supporting Materials

D.1 Personal Records

This section contains personal records in partial support of arguments in the thesis.

***RockyMountainNew.com* Site Inundated on the Morning of a Break in the Ramsey Case** This timed `wget` command was issued on the morning of August 17, 2006, indicating that 2 minutes 38 seconds were required to download the main page from the news server. Note that the content length was *unspecified* (in contrast to the following sample), further indicating that the server was over-utilized and unable to precompute the total content length on HTTP-reply.

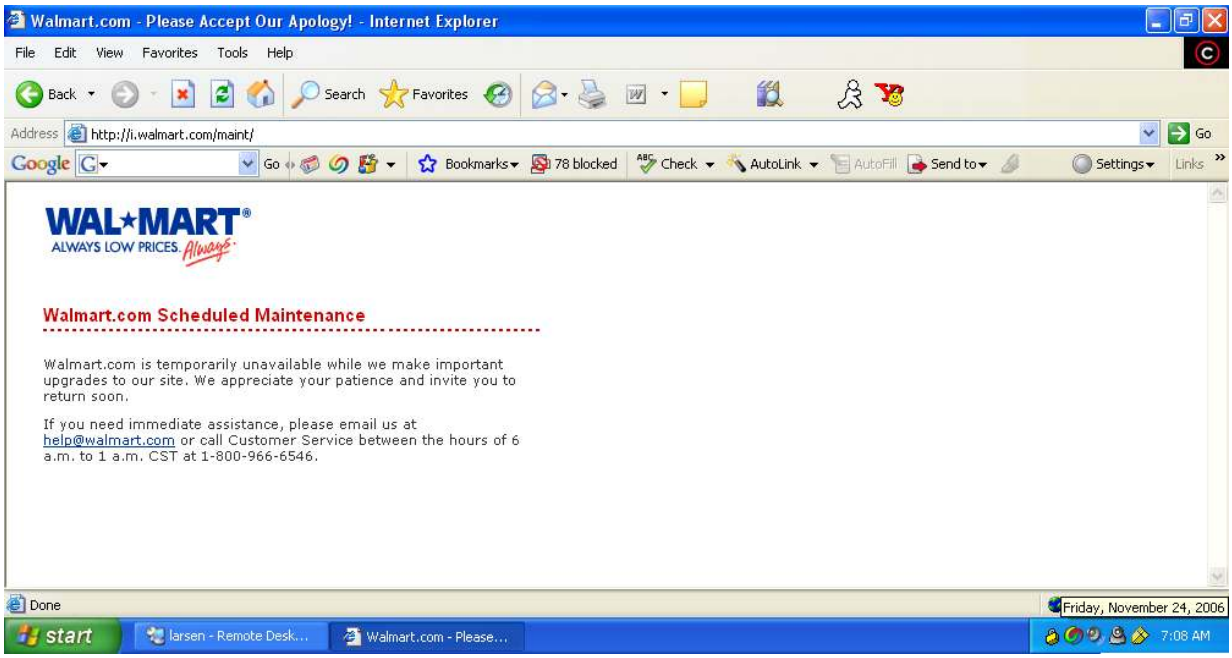
```
$ date
Thu Aug 17 10:18:01 EST 2006

$ time wget http://www.rockymountainnews.com/
--10:18:45--  http://www.rockymountainnews.com/
           => 'index.html'
Resolving www.rockymountainnews.com... 208.62.120.181
Connecting to www.rockymountainnews.com|208.62.120.181|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
    [                                <=> ] 56,309         1.96K/s

10:21:22 (590.01 B/s) - 'index.html' saved [56309]

real    2m37.846s
user    0m0.062s
sys     0m0.218s
```

In contrast, the main page on a typical, low-traffic day downloaded in under two seconds.

```
$ date
Sun Feb 24 14:50:35 EST 2008

$ time wget http://www.rockymountainnews.com/
--14:50:36--  http://www.rockymountainnews.com/
           => 'index.html.2'
Resolving www.rockymountainnews.com... 204.78.38.227
Connecting to www.rockymountainnews.com|204.78.38.227|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 72,061 (70K) [text/html]
```

```
100%[====================================>] 72,061       116.96K/s

14:50:37 (116.68 KB/s) - 'index.html.2' saved [72061/72061]

real    0m1.276s
user    0m0.030s
sys     0m0.062s
```

Wal-Mart Site Outage on Black Friday 2006 This screenshot, courtesy of Marwan Abi-Antoun, captures the outage of Wal-Mart's shopping site on Friday, November 24, 2006.



D.2 Listing and Abstract Descriptions of *netbwe* Subroutines

The five Perl programs and subroutines of the *netbwe* subsystem that provide the core adaptation functionalities are listed below with abstract description of steps.

load_state Loads network usage states by machine from bandwidth logs.

1. Netbwe::Netbwe_db_connect (*operator*) to get DB handle
2. Get list of bandwidth usage logs file from AFS directory, for each...
3. Determine machine state ("State"->status, "Last_Updated_By"->who, "Last_Update"->version+status_date) from state file; status is one of NOTIFIED, WARNED, KILL, EXEMPT; also notice quotaID "1" for wireless, "2" otherwise
4. Grab machine info (%mach) by header pos retrieved with Netbwe::NetReg_lookup: netregID, ip_address, mac_address, users
5. Read each LOG file and process...
6. Log type of: "Created host entry", "Set state to", "Note", "Overwrote lock"; each log entry has version, who, machineID, reason
7. Entries of *machines* and set of *logs* are added to the DB tables of same name, using the logged values (a **db-add** *operator*!)
8. Cleanup: close files and disconnect from DB (*operator*)

```
1  $dbh = CMU::Netbwe::Netbwe_db_connect();
2  open( INLIST, "ls -1 /afs/andrew/acs/ng/etc/bandwidth/*.log | " );
3
4  $mid = 0;
5  foreach $file (<INLIST>) {
6    $mid++;
7    chomp($file);
8    next if ( $file =~ /cmu-yerin/ );
9    $file =~ /(.*)\.log$/;
10   $statefile = $1;
11
12   undef %mach;
13   undef %log;
14
15   open( STATE, $statefile ) || die "Cannot open $statefile\n$!\n";
16   open( LOG,   $file )      || die "Cannot open $file\n$!\n";
17
18   @parts = split( /\//, $statefile );
19   $mach{hostname} = $parts[$#parts];
20
21   while ( $inline = <STATE> ) {
22     chomp $inline;
23     if ( $inline =~ /^State/ ) {
24       $mach{status} = ( split( / /, $inline ) )[1];
25     } elsif ( $inline =~ /^Last_Updated_By/ ) {
26       $mach{who} = ( split( / /, $inline ) )[1];
27     } elsif ( $inline =~ /^Last_Update/ ) {
28       $mach{version} = POSIX::strftime( "%Y%m%d%H%M%S", localtime( ( split( / /,
             $inline ) )[1] ) );
29       $mach{status_date} = POSIX::strftime( "%Y-%m-%d %H:%M:%S", localtime( ( split
             ( / /, $inline ) )[1] ) );
30     }
31   }
32   if ( $mach{status} eq "0" ) {
33     next;
34   } elsif ( $mach{status} eq "1" ) {
35     $mach{status} = 'NOTIFIED';
36   } elsif ( $mach{status} eq "warn" ) {
37     $mach{status} = 'WARNED';
38   } elsif ( $mach{status} eq "kill" ) {
39     $mach{status} = 'KILL';
40   } elsif ( $mach{status} eq 'exempt' ) {
41     $mach{status} = 'EXEMPT';
42   }
43   if ( $mach{hostname} =~ /\.wv\.c[cs]\./ ) {
44     $mach{quotaID} = 1;
45   } else {
46     $mach{quotaID} = 2;
47   }
48
49   $mdata = CMU::Netbwe::NetReg_lookup( "netreg", { HOSTNAME => $mach{hostname} } );
50   if ( ( !ref($mdata) ) || ( $#$mdata == 0 ) ) {
51   } else {
52     $mdpos              = CMU::Netbwe::GetHeaderPos($mdata);
53     $mach{netregID}     = $mdata->[1][ $mdpos->{'machine.id'} ];
54     $mach{ip_address}   = $mdata->[1][ $mdpos->{'machine.ip_address'} ];
55     $mach{mac_address}  = $mdata->[1][ $mdpos->{'machine.mac_address'} ];
56     $mach{users}        = $mdata->[1][ $mdpos->{'internal.users'} ];
57   }
58   $mach{id} = $mid;
59   undef @logs;
60   @logs = ();
61
62   while ( $inline = <LOG> ) {
63     $lent = {};
64     chomp $inline;
```

```
65      ( $etime, $who, $ltype, $text ) = split( /\: /, $inline, 4 );
66
67      if ( $ltype eq "Created host entry" ) {
68        $lent->{version} = POSIX::strftime( "%Y%m%d%H%M%S", localtime( Time::
              ParseDate::parsedate($etime) ) );
69        $lent->{who}       = $who;
70        $lent->{machineID} = $mid;
71        $lent->{reason}    = "$ltype";
72        push( @logs, $lent );
73      } elsif ( $ltype eq "Set state to" ) {
74        $lent->{version} = POSIX::strftime( "%Y%m%d%H%M%S", localtime( Time::
              ParseDate::parsedate($etime) ) );
75        $lent->{who}       = $who;
76        $lent->{machineID} = $mid;
77        if ( $text eq "1" ) {
78          $text = 'NOTIFIED';
79        } elsif ( $text eq "warn" ) {
80          $text = 'WARNED';
81        } elsif ( $text eq "kill" ) {
82          $text = 'KILL';
83        } elsif ( $text eq 'exempt' ) {
84          $text = 'EXEMPT';
85        }
86        $lent->{reason} = "$ltype $text";
87        push( @logs, $lent );
88      } elsif ( $ltype eq "Note" ) {
89        $lent->{version} = POSIX::strftime( "%Y%m%d%H%M%S", localtime( Time::
              ParseDate::parsedate($etime) ) );
90        $lent->{who}       = $who;
91        $lent->{machineID} = $mid;
92        $lent->{reason}    = "$text";
93        push( @logs, $lent );
94      } elsif ( $ltype eq "Overwrote lock" ) {
95        $lent->{version} = POSIX::strftime( "%Y%m%d%H%M%S", localtime( Time::
              ParseDate::parsedate($etime) ) );
96        $lent->{who}       = $who;
97        $lent->{machineID} = $mid;
98        $lent->{reason}    = "$ltype";
99        push( @logs, $lent );
100     } else {
101       die "Don't know how to handle line type $ltype (etime = >$etime<, who = >$who
              <, ltype = >$ltype<, text = >$text<)\n";
102     }
103   }
104
105   ( $add, $reason ) = Add( $dbh, "machines", \%mach );
106   if ($add) {
107     die "Could not add entry " . join( ", ", @$reason ) . "\n";
108   }
109   map {
110     ( $add, $reason ) = Add( $dbh, "logs", $_ );
111     if ($add) {
112       die "Could not add entry " . join( ", ", @$reason ) . "\n";
113     }
114   } (@logs);
115
116   close(LOG);
117   close(STATE);
118 }
119 $dbh->disconnect();
120 exit(0);
```

log_usage Determines and records current usage against database of prior machine states.

1. Target a host for which to log usage
2. Netbwe::Netbwe_db_connect to get DB handle
3. Fetch list of sensors via Netbwe::sensor_list, and for each sensor... collect its set of info to generate commands
4. Issue command via “/usr/ng/bin/topt” to collect log messages
5. Parse message by either: hostname+IP-address or MAC+Hostname (a **parse-message** *operator*!)
6. For each message, fetch values: outboundRate, inboundRate, eventdate, sensorID, ip_address/mac_address+subnet
7. Call Netbwe::API::event_add to add the fetched message values by user ’netbwe’ to ’events’ table
8. Cleanup: disconnect from DB

```
1  my ($host) = `/bin/hostname`;
2  chomp $host;
3  $dbh = CMU::Netbwe::Netbwe_db_connect();
4  if ( !$dbh ) {
5    die __FILE__ . ":" . __LINE__ . ": Couldn't open database\n";
6  }
7  my $sensors = CMU::Netbwe::sensor_list( $dbh, 'netbwe', "host_name like '$host'" );
8  if ( ref $sensors ) {
9    my $dapos = CMU::Netbwe::GetHeaderPos($sensors);
10   shift @$sensors;
11   foreach my $sensor (@$sensors) {
12     my ($comopts) = {
13               $host => {
14                 opts          => $sensor->[ $dapos->{'sensors.options'} ],
15                 file          => $sensor->[ $dapos->{'sensors.file'} ],
16                 raw_data_path => $sensor->[ $dapos->{'sensors.raw_data_path'} ],
17                 prog          => "/usr/ng/bin/topt",
18               }
19     };
20     if (    ( !defined $comopts->{$host}{opts} )
21          || ( !defined $comopts->{$host}{file} ) ) {
22       die "No configuration is defined for this sensor.($sensor->[$dapos->{'sensors
           .name'}])\n";
23     }
24     if ( defined $opts{i} ) {
25       open( INFILE, $opts{i} ) || die "Could not open $opts{i} for read\n$!\n";
26       $infile = \*INFILE;
27     } else {
28       $infile = \*STDIN;
29     }
30     if ( defined $opts{o} ) {
31       open( OUTFILE, ">$opts{o}" )
32         || die "Could not open $opts{o} for write\n$!\n";
33       $outfile = \*OUTFILE;
34     } else {
35       $outfile = \*STDOUT;
36     }
37     if ( defined $opts{D} ) {
38       $date = $opts{D};
39       if ( $date eq 'today' ) {
40         $date = $date = strftime "%Y/%m/%d", localtime;
41       } elsif ( $date eq 'yesterday' ) {
42         $date = $date = strftime "%Y/%m/%d",
43           localtime( Time::ParseDate::parsedate('yesterday') );
44       } elsif ( $date !~ /^20\d\d\/\d\d?\/\d\d?$/ ) {
45         die
46 "Invalid date format, use 'YYYY/MM/DD', 'today' or 'yesterday' only\n";
47       }
48     } else {
49       $date = $date = strftime "%Y/%m/%d",
50         localtime( Time::ParseDate::parsedate('yesterday') );
51     }
```

```
52
53      # make sure raw_data_path has a default value:
54      unless ( $comopts->{$host}{raw_data_path} ) {
55        $comopts->{$host}{raw_data_path} = '/home/argus/archive';
56      }
57      $command = "$comopts->{$host}{prog} $comopts->{$host}{opts} "
58        . "$comopts->{$host}{raw_data_path}/$date/.counts/$comopts->{$host}{file}";
59      $msg = join( "", `$command` );
60      $msg =~ s/\r//sg;
61      $msg =~ s/IP address/IP-address/sg;
62      if ( $msg =~ /IP-address/ ) {
63        $parsed = ParseMessage( $msg, [ 'Hostname', 'IP-address' ] );
64      } elsif ( $msg =~ /MAC       /s ) {
65        $parsed = ParseMessage( $msg, [ 'MAC', 'Hostname' ] );
66      } else {
67        $parsed = undef;
68      }
69      if ( !$parsed ) {
70        die __FILE__ . ":" . __LINE__ . ": Parse failed, message was something like \
              n" . Data::Dumper->Dump( [$msg], ['msg'] ) . "\n";
71      }
72      $papos = CMU::Netbwe::GetHeaderPos($parsed);
73      shift @$parsed;
74
75      $skipped_ignore = 0;
76      $skipped_netreg = 0;
77      $skipped_pool   = 0;
78      $skipped_dup    = 0;
79      foreach $j (@$parsed) {
80        undef $vars;
81        $vars->{outboundRate} = $j->[ $papos->{oMB} ];
82        $vars->{inboundRate} = $j->[ $papos->{iMB} ];
83        $vars->{eventdate} = $date;
84        $vars->{sensorID}  = $sensor->[ $dapos->{'sensors.id'} ];
85
86        if ( $papos->{Hostname} == 0 )
87        {    # this one came from a probe that logs by IP
88          $vars->{ip_address} = $j->[ $papos->{'IP-address'} ];
89        } else {    # this one came from a probe that logs by mac address
90          $vars->{mac_address} = $j->[ $papos->{MAC} ];
91          $vars->{subnet} = $comopts->{$host}{subnet}
92            if ( defined $comopts->{$host}{subnet} );
93        }
94
95        ( $val, $reason ) = CMU::Netbwe::event_add( $dbh, 'netbwe', $vars );
96        if ( !ref($val) ) {
97          if ( $val == $CMU::Netbwe::errvals->{ENOENT} ) {
98            $skipped_netreg++;
99          } elsif ( $val == $CMU::Netbwe::errvals->{EIGNORE} ) {
100           $skipped_ignore++;
101         } elsif ( $val == $CMU::Netbwe::errvals->{EPOOL} ) {
102           $skipped_pool++;
103         } elsif (scalar grep { $_ =~ /Duplicate/ } @$reason ) {
104           $skipped_dup++;
105           next;
106         } else {
107           die __FILE__ . ":" . __LINE__ . ": $CMU::Netbwe::errmeanings->{$val}: [ "
                  . join( ', ', @$reason ) . " ]\n";
108         }
109       }
110     }
111     $skipped_total = $skipped_ignore + $skipped_netreg + $skipped_pool +
            $skipped_dup;
112   }
113 }
114 $dbh->disconnect();
```

do_violation Determines violation incidents per machine, sends notifications to machine owners, records violation "state" change.

1. $commit = 'yes', $mail = 'yes'; fetch list of dates for which to process violation
2. Netbwe::Netbwe_db_connect to get DB handle
3. Netbwe::quota_list to fetch list of quota categories by user 'netbwe' from 'quotas' table
4. For each quota category $qname(quota), for each $day(date) specified, for each $mode "daily" and "initial"...
   (a) process violations (see API::violations) and collect list of hosts whose states reflect bandwidth usage violations
5. Generate an email (*operator*) to the "Abuse BBoard" containing the list of state-changed hosts, incl. hostname+ip_address+mac_address+status, sorted by status
6. Cleanup: disconnect from DB and sendmail

```
1  $commit = 'yes';
2  $mail   = 'yes';
3  $user = "netbwe";
4  if ( defined $opts{i} ) {
5    open( INFILE, $opts{i} ) || die "Could not open $opts{i} for read\n$!\n";
6    $infile = \*INFILE;
7  } else {
8    $infile = \*STDIN;
9  }
10 if ( defined $opts{o} ) {
11   open( OUTFILE, ">$opts{o}" ) || die "Could not open $opts{o} for write\n$!\n";
12   $outfile = \*OUTFILE;
13 } else {
14   $outfile = \*STDOUT;
15 }
16 if ( defined $opts{D} ) {
17   push( @days, split( /,/, $opts{D} ) );
18 } else {
19   push( @days, POSIX::strftime( "%Y-%m-%d", localtime( time() - 86400 ) ) );
20 }
21
22 $dbh = CMU::Netbwe::Netbwe_db_connect();
23 @hosts = ();
24 $quotas = CMU::Netbwe::quota_list( $dbh, 'netbwe', 'disable = "NO"' );
25 if ( ref $quotas ) {
26   my $quota_pos = CMU::Netbwe::GetHeaderPos($quotas);
27   shift @$quotas;
28   foreach $quota (@$quotas) {
29     my $qname = $quota->[ $quota_pos->{'quotas.name'} ];
30     foreach $day (@days) {
31       foreach $mode (qw/daily initial/) {
32         ( $data, $reason ) = CMU::Netbwe::violations( $dbh, $user, $qname, $day,
               $mode, $commit, $mail, $qlookup );
33         die __FILE__ . ":" . __LINE__ . "$CMU::Netbwe::errmeanings->{$data} [" .
               join( ', ', $reason ) . "] \n" if ( ( !ref $data ) && ( $data != 0 ) );
34         if (@$data) {
35           shift @$data if (@hosts);
36           push( @hosts, @$data );
37         }
38       }
39     }
40   }
41 }
42
43 $hopos = CMU::Netbwe::GetHeaderPos( \@hosts );
44 $msg = "";
45 $msg .= "X-Mailer: Network Bandwidth Monitoring System\n";
46 $msg .= "Auto-Submitted: yes\n";
47 $msg .= "Reply-To: abuse-bandwidth\@andrew.cmu.edu\n";
```

```
48 $msg .= "From: Bandwidth Usage <abuse-bandwidth\@andrew.cmu.edu>\n";
49 $msg .= "To: \"Abuse Bboard\" <post+org.acs.ng.abuse\@andrew.cmu.edu>\n";
50 $msg .= "Date: " . CMU::Netdb::ArpaDate() . "\n";
51 $msg .= "Subject: Bandwidth State Changes for $days[0]\n";
52 $msg .= "\n";
53 $msg .= "State Changes due to high bandwith usage/expiration\n\n";
54 shift @hosts;
55 @hosts = sort {
56   $a->[ $hopos->{'machines.status'} ] cmp $b->[ $hopos->{'machines.status'} ]
57 } @hosts;
58 foreach (@hosts) {
59   $msg .= swrite(
60 <<'END', $_->[ $hopos->{'machines.hostname'} ], CMU::Netbwe::long2dot( $_->[ $hopos
       ->{'machines.ip_address'} ] ), $_->[ $hopos->{'machines.mac_address'} ], $_->[
       $hopos->{'machines.status'} ] );
61 @<<<<<<<<<<<<<<<<<< | @<<<<<<<<<<<<<< | @<<<<<<<<<< | @<<<<<<<<<<<<<<<<<<
62 END
63 }
64 $dbh->disconnect();
65 CMU::Netbwe::send_mail( $msg, 'post+org.acs.ng.abuse@andrew.cmu.edu', undef, 'post+
       org.acs.ng.project.bandwidth@andrew.cmu.edu' );
66 exit(0);
```

Netbwe::API::violations ($dbh, $user, $quota, $date, $mode, $commit='no', $mail='no', $qlookup) collects list of hosts whose states reflect bandwidth usage violations.

1. Netbwe::helper::chkParam (*operator*); in general, Netbwe::helper and Netbwe::primitives are candidate operators
2. Check and standardize the parameters:
   (a) $date: Netbwe::helper::TimeStamp to convert an Int or parse a string
   (b) $quota: fetch identified quota, populate quota.id($quota), sensorID($sensor), name($qname), type($quota_type), initialPeriod($days), initialQuota($limit), dailyQuota($dlimit)
   (c) $mode: make sure it's either "daily" or "initial"
   (d) $commit & $mail: "yes" or "no"
3. Fetch list of events ('events' table) using Netbwe::API::event_list (*operator*) for a specific date if "daily" and the initialPeriod if "initial", for each event...
   (a) Aggregate bandwidth usage of event per machine ($mach->{$hostkey})
   (b) $hostkey is based on one of netregID ('netreg:'), mac_address ('hwaddr:'), or ip_address ('ipaddr:')
   (c) Store 'rates' per date for 'inbound' & 'outbound'
   (d) Accumulate both 'outboundRate' & 'inboundRate'
   (e) Accumulate 'Rate' based on $quota_type
      i. 'outboundRate' if "Outbound"
      ii. 'inboundRate' if "Inbound"
      iii. Max of in & out if "Either"
      iv. Sum of in & out if "Total"
4. For each machine iden'd by $key in %$mach...
   (a) Drop machine if 'Rate' is below initialQuota($limit)
   (b) Determine if machine status is "exempted" using Netbwe::helper::ChkStatus (*operator*)
   (c) Collect the log entry value set from $mach->{$key}{...}, incl. 'quotaID' assigned with $quota
   (d) If $commit=='yes', invoke API::log_violation (*operator*) to determine if any violation has occurred and log to netnotify
   (e) If there's a logged violation for machine, store it for machine in 'result'
5. Collect and return array of machine violations

```
1  sub violations {
2    my ( $dbh, $user, $quota, $date, $mode, $commit, $mail, $qlookup ) = @_;
3    my ( $data, $dapos, $reason );
4    my ( $days, $limit, $dlimit, $mach, $quota_type, $sensor );
5    my ( $vio, $vals, $val, $retval, $qname );
6    $commit = 'no' if ( !defined $commit );
7    $mail   = 'no' if ( !defined $mail );
8
9    # check/standardize date param
10   if ( CMU::Netbwe::chkParam( \$date, ['DATE_TIME'] ) ) {
11     if ( CMU::Netbwe::chkParam( \$date, ['INTEGER'] ) ) {
12       return (wantarray ? ($errvals->{EINVALID}, ['date']) : $errvals->{EINVALID});
13     } else {
14       $date = CMU::Netbwe::TimeStamp($date);
15     }
16   } else {
17     $date = CMU::Netbwe::TimeStamp( Time::ParseDate::parsedate($date) );
18   }
19   # check and get quota information
20   if ( CMU::Netbwe::chkParam( \$quota, ['INTEGER'] ) ) {
21     if ( CMU::Netbwe::chkParam( \$quota, ['QUERY_CLEAN'] ) ) {
22       return (wantarray ? ($errvals->{EINVALID},['quota']) : $errvals->{EINVALID});
23     }
24     ($data,$reason)=CMU::Netbwe::quota_list($dbh,$user,"(quotas.name=\"$quota\")");
25   } else {
26     ($data,$reason)=CMU::Netbwe::quota_list($dbh,$user,"(quotas.id=\"$quota\")");
27   }
28   if ( !ref $data ) {
29     return (wantarray ? ($data, $reason) : $data);
30   } elsif ( $#$data == 0 ) {
31     return (wantarray ? ($errvals->{ENOENT}, ['quota']) : $errvals->{EINVALID});
32   }
33   $dapos      = CMU::Netbwe::GetHeaderPos($data);
34   $quota      = $data->[1][ $dapos->{'quotas.id'} ];
35   $sensor     = $data->[1][ $dapos->{'quotas.sensorID'} ];
36   $qname      = $data->[1][ $dapos->{'quotas.name'} ];
37   $quota_type = $data->[1][ $dapos->{'quotas.type'} ];
38   $days       = $data->[1][ $dapos->{'quotas.initialPeriod'} ];
39   $limit      = $data->[1][ $dapos->{'quotas.initialQuota'} ];
40   $dlimit     = $data->[1][ $dapos->{'quotas.dailyQuota'} ];
41
42   # Check mode parameter
43   if ( CMU::Netbwe::chkParam( \$mode, [ 'ONE_OF', [ 'daily', 'initial' ] ] ) ) {
44     return (wantarray ? ($errvals->{EINVALID}, ['mode']) : $errvals->{EINVALID});
45   }
46   # Check commit parameter
47   if ( CMU::Netbwe::chkParam( \$commit, [ 'ONE_OF', [ 'yes', 'no' ] ] ) ) {
48     return (wantarray ? ($errvals->{EINVALID}, ['commit']) : $errvals->{EINVALID});
49   }
50   # check mail parameter (obsoleted by NetNotify, mail always sent)
51   if ( CMU::Netbwe::chkParam( \$mail, [ 'ONE_OF', [ 'yes', 'no' ] ] ) ) {
52     return (wantarray ? ($errvals->{EINVALID}, ['mail']) : $errvals->{EINVALID});
53   }
54
55   # Get event list based on mode
56   if ( $mode eq 'daily' ) {
57     $limit = $dlimit;
58     ( $data, $reason ) = event_list( $dbh, $user, "((events.sensorID = \"$sensor\")
           and (events.eventdate = \"$date\"))", ['events.eventdate'] );
59   } else {
60     my $interval = $days - 1;
61     ( $data, $reason ) = event_list( $dbh, $user, "((events.sensorID = \"$sensor\")
           and (events.eventdate between (\"$date\" - interval $interval day) and \"
         $date\"))", ['events.eventdate'] );
62   }
63   if ( !ref $data ) {
```

```
64      return ( wantarray ? ( $data, $reason ) : $data );
65    }
66    $dapos = CMU::Netbwe::GetHeaderPos($data);
67    shift @$data;
68
69    # Add up the events, based on netregID, mac_address, or IP address
70    foreach my $row (@$data) {
71      my $hostkey = undef;
72      if ( defined $row->[ $dapos->{'events.netregID'} ]
73           && $row->[ $dapos->{'events.netregID'} ] != 0 )
74      {
75        # If the event has a netregID, add them up by that.
76        # never include the IP if we've got the netreg ID
77        $hostkey = "netreg:" . $row->[ $dapos->{'events.netregID'} ];
78        $mach->{$hostkey}{netregID}     = $row->[ $dapos->{'events.netregID'} ];
79        $mach->{$hostkey}{mac_address} = $row->[ $dapos->{'events.mac_address'} ]
80          if ( defined $row->[ $dapos->{'events.mac_address'} ]
81               && $row->[ $dapos->{'events.mac_address'} ] ne "" );
82      } elsif ( defined $row->[ $dapos->{'events.mac_address'} ]
83                && $row->[ $dapos->{'events.mac_address'} ] ne "" )
84      {
85        # Or, if the event has a mac address, add them up that way.
86        # (I think having a mac addr, but not having a netreg ID should never happen,
              but its here for completeness)
87        $hostkey = "hwaddr:" . $row->[ $dapos->{'events.mac_address'} ];
88        $mach->{$hostkey}{ip_address} =
89          CMU::Netbwe::long2dot( $row->[ $dapos->{'events.ip_address'} ] )
90          if ( defined $row->[ $dapos->{'events.ip_address'} ]
91               && $row->[ $dapos->{'events.ip_address'} ] != 0 );
92        $mach->{$hostkey}{mac_address} = $row->[ $dapos->{'events.mac_address'} ];
93      } elsif ( defined $row->[ $dapos->{'events.ip_address'} ]
94                && $row->[ $dapos->{'events.ip_address'} ] != 0 )
95      {
96        # Finally, if we have no netregID or mac, it must be from one of the sub
              networks (CS, ECE, SEI), and thus we
97        # only have an IP and hostname
98        $hostkey = "ipaddr:" . $row->[ $dapos->{'events.ip_address'} ];
99        $mach->{$hostkey}{ip_address} =
100         CMU::Netbwe::long2dot( $row->[ $dapos->{'events.ip_address'} ] );
101       $mach->{$hostkey}{hostname} = $row->[ $dapos->{'events.host_name'} ]
102         if ( defined $row->[ $dapos->{'events.host_name'} ]
103              && $row->[ $dapos->{'events.host_name'} ] ne "" );
104     } else {
105       next;
106     }
107
108     $mach->{$hostkey}{'rates'}{ $row->[ $dapos->{'events.eventdate'} ] }
109       {'inbound'} = $row->[ $dapos->{'events.inboundRate'} ];
110     $mach->{$hostkey}{'rates'}{ $row->[ $dapos->{'events.eventdate'} ] }
111       {'outbound'} = $row->[ $dapos->{'events.outboundRate'} ];
112     $mach->{$hostkey}{outboundRate} =
113       ( $mach->{$hostkey}{outboundRate} || 0 ) + $row->[ $dapos->{'events.
              outboundRate'} ];
114     $mach->{$hostkey}{inboundRate} =
115       ( $mach->{$hostkey}{inboundRate} || 0 ) + $row->[ $dapos->{'events.
              inboundRate'} ];
116     if ( $quota_type eq 'Outbound' ) {
117       $mach->{$hostkey}{Rate} = ( $mach->{$hostkey}{Rate} || 0 ) + $row->[ $dapos
              ->{'events.outboundRate'} ];
118     } elsif ( $quota_type eq 'Inbound' ) {
119       $mach->{$hostkey}{Rate} = ( $mach->{$hostkey}{Rate} || 0 ) + $row->[ $dapos
              ->{'events.inboundRate'} ];
120     } elsif ( $quota_type eq 'Either' ) {
121       $mach->{$hostkey}{Rate} = ( $mach->{$hostkey}{Rate} || 0 ) +
122         CMU::Netbwe::max( $row->[ $dapos->{'events.outboundRate'} ],
123                           $row->[ $dapos->{'events.inboundRate'} ] );
```

```
124       } elsif ( $quota_type eq "Total" ) {
125         $mach->{$hostkey}{Rate} =
126           ( $mach->{$hostkey}{Rate} || 0 ) + $row->[ $dapos->{'events.outboundRate'}
                ] + $row->[ $dapos->{'events.inboundRate'} ];
127       }
128     }
129
130     # Process the list
131     foreach my $key ( keys %$mach ) {
132       # flush out the ones that are not over quota
133       if ( $mach->{$key}{Rate} <= $limit ) {
134         delete $mach->{$key};
135         next;
136       }
137       # Drop any machine that is exempt (****)
138       my $query;
139       $query->{mac_address} = $mach->{$key}{mac_address}
140         if ( defined $mach->{$key}{mac_address} );
141       $query->{netregID} = $mach->{$key}{netregID}
142         if ( defined $mach->{$key}{netregID} );
143       $query->{ip_address} = $mach->{$key}{ip_address}
144         if ( !defined $query->{netregID} && defined $mach->{$key}{ip_address} );
145       my ($status, $reason, $extra) = CMU::Netbwe::ChkStatus($dbh, $user, $query,
            $qlookup);
146       if ( $status eq "exempted" ) {
147         delete $mach->{$key};
148         next;
149       } elsif ( $status == $CMU::Netbwe::errvals->{EMULTIPLE} ) {
150         # ChkStatus returned the extra bits, iterate over them
151         if ( ref $extra ) {
152           # yes, just look at that data
153           foreach my $m (@$extra) {
154             if ( $m eq "exempted" ) {
155               delete $mach->{$key};
156               last;
157             }
158           }
159         }
160         if ( !defined $mach->{$key} ) {
161           delete $mach->{$key};
162           next;
163         }
164       }
165
166       # prep the log entry val set
167       undef $vals;
168       $vals->{quotaID} = $quota;
169       foreach $val ( keys %{ $mach->{$key} } ) {
170         $vals->{$val} = $mach->{$key}{$val} if ( !( $val =~ /Rate/ ) );
171       }
172       # if commit is yes, then log the violation to netnotify
173       ( $vio, $reason ) = log_violation(
174                           $dbh,                          $user,
175                           $vals,                         $mach->{$key}{outboundRate},
176                           $mach->{$key}{inboundRate}, $quota,
177                           $date,                         $mode,
178                           $mail,                         $limit,
179                           $qlookup,                      $mach->{$key}{'rates'},
180                           $mach->{$key}{'Rate'}
181         )
182         if ( $commit eq 'yes' );
183       if (
184               ( defined $vio )
185           && ( !ref $vio )
186           && (     ( $vio == $CMU::Netbwe::errvals->{EXEMPT_MACHINE} )
187                 || ( $vio == $CMU::Netbwe::errvals->{ENTRY_EXISTS} )
```

```
188                 || ( $vio == $CMU::Netbwe::errvals->{EPREMATURE} )
189                 || ( $vio == $CMU::Netbwe::errvals->{ENOSTATE} ) )
190         )
191       {
192         delete $mach->{$key};
193         next;
194       }
195       if ( ( defined $vio ) && ( ref $vio ) ) {
196         $mach->{$key}{result} = $vio;
197       }
198     }
199
200     @$retval = ();
201     foreach ( keys %$mach ) {
202       next unless ( defined $mach->{$_}{result} );
203       $data = $mach->{$_}{result};
204       if (@$retval) {
205         shift(@$data);
206         push( @$retval, @$data );
207       } else {
208         @$retval = @$data;
209       }
210     }
211     return ( wantarray ? ( $retval, "" ) : $retval );
212   }
```

Netbwe::API::log_violation ($dbh, $user, $svals, $outbound, $inbound, $quota, $date, $mode, $mail, $limit, $qlookup, $rates, $totalRate) determines if violation occurred & logs to *netnotify*.

1. Use API::SOAP_machine_list (*operator*) to search DB for existing violation record of current machine + quota category pairing
2. If $mode is "initial"
   (a) If violation record found, then do no more and fail with ENTRY_EXISTS since the "daily" mode will take care of it
   (b) Otherwise, create a new entry with: machines.mac_address/ip_address/netregID, machines.status, machines.status_date, machines.reason, machines.quotaID, OUTBOUND_USAGE, INBOUND_USAGE, AVERAGE_RATE, and 'RATES' (which lists inbound & outbound usage by date)
   (c) Invoke API::machine_add (*operator*?!) to log the violation incident on Epidemic via SOAP
3. If $mode is "daily"
   (a) If not existing violation record, nothing to do; otherwise...
   (b) Check on exempt status, and return with error EXEMPT_MACHINE
   (c) Call API::full_state_list_hash (*operator*) to obtain list of predefined state changes for this quota category
   (d) Determine $next_state by referencing list of states to see what next state to "promoteto" given current machine state
   (e) If there's no next state, bail with ENOSTATE
   (f) Otherwise, make the "state transition" with a modification to the machine violation entry
      i. New status data: machines.status, machines.status_date, machines.reason, OUTBOUND_USAGE, INBOUND_USAGE (no RATES list since this is a daily run)
      ii. Invoke API::machine_modify (*operator*) to transition the state of the violation incident on Epidemic via SOAP
4. Otherwise bail with EINVALID

```
1  sub log_violation {
2    my ( $dbh,     $user,  $svals, $outbound, $inbound,
3         $quota,   $date,  $mode,  $mail,     $limit,
4         $qlookup, $rates, $totalRate
5       ) = @_;
```

```
6     my ( %vals,    $event_time );
7     my ( $data,    $dapos, $reason );
8     my ( $ttimes, $next_state );
9
10    # Check for old record
11    ( $data, $reason ) = SOAP_machine_list( $dbh, $user, $svals, undef, undef,
          $qlookup, 1 );
12    if ( !ref $data ) {
13      return ( wantarray ? ( $data, $reason ) : $data );
14    } elsif ( $#$data > 1 ) {
15      # We have a problem, we got multiple rows back on what should be a unique key.
16      $reason = ["Multiple rows for machine/quota found"];
17      return ( wantarray ? ( $CMU::Netbwe::errvals->{EMULTIPLE}, $reason ) : $data );
18    }
19    $dapos = CMU::Netbwe::GetHeaderPos($data) if ( ref $data );
20    # Is this the initial finding of this host (ie a n day total overusage)
21    if ( $mode eq 'initial' ) {
22      if ( $#$data == 1 ) {
23        # If already in a state, they will get promoted by the daily, so just return;
24        return ( wantarray ? ( $CMU::Netbwe::errvals->{ENTRY_EXISTS}, ["Entry Exists"
              ] ) : $CMU::Netbwe::errvals->{ENTRY_EXISTS} );
25      }
26      # log a new entry for them...
27      $vals{'machines.mac_address'} = $svals->{'machines.mac_address'}
28        if ( defined $svals->{'machines.mac_address'} );
29      $vals{'machines.mac_address'} = $svals->{mac_address}
30        if ( defined $svals->{mac_address} );
31      $vals{'machines.ip_address'} = $svals->{ip_address}
32        if ( defined $svals->{ip_address} );
33      $vals{'machines.netregID'} = $svals->{netregID}
34        if ( defined $svals->{netregID} );
35      $vals{'machines.status'} = 'initial-notification-graceperiod';
36      $vals{'machines.status_date'} = $date;
37      $vals{'machines.reason'} = "Exceeded $mode quota (transmitted $outbound Mbytes
            outbound, $inbound Mbytes inbound) state $vals{machines.status} on $date";
38      $vals{'machines.quotaID'} = $quota;
39      $vals{'OUTBOUND_USAGE'}   = sprintf( "%-8.3f", $outbound / 1024 );
40      $vals{'INBOUND_USAGE'}    = sprintf( "%-8.3f", $inbound / 1024 );
41      $^A                       = "";
42      formline("             @>>>>>>>>>    @>>>>>>>>>\n", 'Inbound', 'Outbound');
43      formline("@>>>>>>>>>   @>>>>>>>>>    @>>>>>>>>>\n", 'Date', 'Usage', 'Usage');
44      $vals{'AVERAGE_RATE'} = sprintf( "%.3f", $totalRate / 1024 / 5 );
45      foreach my $ratedate ( sort keys %$rates ) {
46        my ($date) = split( ' ',      $ratedate );
47        my $in     = sprintf( "%.3f", $rates->{$ratedate}{'inbound'} / 1024 );
48        my $out    = sprintf( "%.3f", $rates->{$ratedate}{'outbound'} / 1024 );
49        if ( $in > $out ) {
50          formline("@>>>>>>>>> @>>>>>>>>>GB* @>>>>>>>>>GB \n", $date, $in, $out);
51        } else {
52          formline("@>>>>>>>>> @>>>>>>>>>GB  @>>>>>>>>>GB*\n", $date, $in, $out);
53        }
54      }
55      $vals{'RATES'} = $^A;
56      ( $data, $reason ) = machine_add( $dbh, $user, \%vals, $mail );
57
58      if ( !ref $data ) {
59        return ( wantarray ? ( $data, $reason ) : $data );
60      }
61    } elsif ( $mode eq 'daily' ) {
62      if ( $#$data == 0 ) {
63        # daily run, only smack people with an existing entry.
64        return ( wantarray ? ( 0, undef ) : 0 );
65      }
66      # check to see if machine is exempt, if so, just return with exempt error
67      if ( $data->[1][ $dapos->{'machines.status'} ] eq "exempted" ) {
68        return ( wantarray
```

```
69                  ? ( $CMU::Netbwe::errvals->{EXEMPT_MACHINE}, ["Exempt"] )
70                  : $CMU::Netbwe::errvals->{EXEMPT_MACHINE} );
71      }
72      #  Get the list of state change times for this quota
73      ( $ttimes, $reason ) = full_state_list_hash( $dbh, $user, $quota );
74      if ( !ref $ttimes ) {
75        return ( wantarray ? ( $data, $reason ) : $data );
76      }
77      $next_state = $ttimes->{$data->[1][$dapos->{'machines.status'}]}{promoteto};
78      if ( defined $next_state ) {
79        # If we have a defined transition, set up to make it.
80        $vals{'machines.status'} = $ttimes->{ $data->[1][ $dapos->{'machines.status'}
              ] }{promoteto};
81        $vals{'machines.status_date'} = $date;
82        $vals{'machines.reason'} = "Exceeded $mode quota (transmitted $outbound
              Mbytes outbound, $inbound Mbytes inbound) state $vals{'machines.status'}
              on $date";
83        $vals{'OUTBOUND_USAGE'} = sprintf( "%-8.3f", $outbound / 1024 );
84        $vals{'INBOUND_USAGE'}  = sprintf( "%-8.3f", $inbound / 1024 );
85
86        # if there is no transition time from the current state, return
87        if ( ( !defined($ttimes->{ $data->[1][ $dapos->{'machines.status'} ] }{
              promote}) ) || ( $ttimes->{ $data->[1][ $dapos->{'machines.status'} ] }{
              promote} == 0 ) ) {
88          return ( wantarray ? ( $CMU::Netbwe::errvals->{ENOSTATE}, ["No Next State"]
                ) : $CMU::Netbwe::errvals->{ENOSTATE} );
89        } else {
90          ( $data, $reason ) = machine_modify( $dbh, $user, $data->[1][ $dapos->{'
                machines.id'} ], $data->[1][ $dapos->{'machines.version'} ], \%vals,
                $mail );
91          if ( !ref $data ) {
92            return ( wantarray ? ( $data, $reason ) : $data );
93          }
94        }
95      } else {
96        return ( wantarray ? ( $CMU::Netbwe::errvals->{ENOSTATE}, ["No Next State"] )
              : $CMU::Netbwe::errvals->{ENOSTATE} );
97      }
98    } else {
99      return ( wantarray ? ( $CMU::Netbwe::errvals->{EINVALID}, ["mode"] ) : $CMU::
          Netbwe::errvals->{EINVALID} );
100   }
101   return ( wantarray ? ( $data, $reason ) : $data );
102 }
```