Making Self-adaptation an Engineering Reality

Shang-Wen Cheng, David Garlan, and Bradley Schmerl

School of Computer Science,
Carnegie Mellon University,
5000 Forbes Ave, Pittsburgh, PA 15213
{zensoul, garlan, schmerl}+@cs.cmu.edu

Abstract. In this paper, we envision a world where a software engineer could take an existing software system, specify, for a set of properties of interest, an objective, conditions for change, and strategies for their adaptation and, within a few man weeks, make that system self-adaptive where it was not before. We describe how our approach generalizes to different classes of systems and holds promise for cost-effective, dynamic system self-adaptation to become an engineering reality.

1 Introduction

Imagine a world where a software engineer could take an existing software system, specify for a set of properties of interest, an objective, conditions for change, and strategies for their adaptation and, within a few man-weeks, make that system self-adaptive where it was not before. An engineer might take an existing client-server system and make it self-adaptive with respect to a specific performance concern such as high latency. He might specify an objective to maintain request-response latency below some threshold, a condition to change the system if the latency rises above the threshold, and a few strategies to adapt the system to fix the high-latency situation. Another engineer might make a coalition-of-services system self-adaptive to network performance fluctuations while keeping down cost of operating the infrastructure. Still another engineer might make a cluster of servers self-adaptive to certain security attacks.

Systems with mechanisms to monitor and adapt themselves to faults or surrounding changes are known variously as *self-adaptive*, *self-healing*, or *self-managing* systems. A decade in the past, systems that supported self-adaptation were rare, confined mostly to domains like telecommunications switches or deep space control software, where shutdown for upgrades was not an option and human intervention was not always possible.

Today, more and more systems have this requirement. 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 errors (e.g., server components failing, or connections going down), and changing user priorities (e.g., high-fidelity video streams at one moment and low fidelity at another). Ubiquitous computing, in which

O. Babaoglu et al. (Eds.): SELF-STAR 2004, LNCS 3460, pp. 158 – 173, 2005.

highly mobile users operate in heterogeneous environments under resource constraints, also motivates the need for self-adaptive systems. Finally, leading software companies like IBM [11] are pursuing ways to develop "self-managing and self-provisioning" infrastructure to help businesses streamline IT operations [10].

Over the past decade, 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 retrofit systems with the capabilities. Moreover, once added, the capabilities are difficult to modify and usually provide only localized treatment of system errors [14,27].

How might we achieve the kind of envisioned capabilities for self-adaptation? Clearly there are many lines of research that must contribute, including (a) new mechanisms for monitoring the behavior of systems in order to detect when problems occur; (b) new techniques for diagnosing and correcting problems once they are detected; and (c) new capabilities for run-time reconfiguration that will support on-line adaptation. However, even if these capabilities were somehow magically available, there would still remain the important problem of making it possible for engineers to use them in cost-effective and principled ways. In particular, we would like to be sure that engineers can augment existing systems to be self-adaptive without having to rewrite them from scratch, that self-adaptation policies and strategies can be used 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 previous work, we have developed a framework incorporating some of the mechanisms mentioned above and demonstrated end-to-end self-adaptation support through two case studies [4,12,13]. We have also described the use of software architectural style to support analysis and guide decisions for system monitoring, diagnosis, and changes [3]. In this work, we show how our approach generalizes across different classes of systems, and re-examine in this context our existing case studies as well as a new case study on security concern.

2 Related Work

Our work builds on a rich set of existing technologies for dynamic system adaptation, and improves upon a number of prior approaches.

2.1 Technologies for Dynamic System Adaptation

Gross and colleagues at Columbia University have contributed substantial work on monitoring—probing and gauging—and effecting technologies [19,30]. The DASADA project has defined the probe and gauge infrastructures [1,15]. Event systems like SIENA [2] and MEET [18] provide the communication infrastructure necessary for monitoring. Workflow systems have been applied to support planning in self-adaptation, such as the Cougaar-based self-adaptation by BBN Technologies [6].

A similar body of research applies adaptation 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 [9,22]. 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 [20]. Adaptive middleware technology may prove synergistic with our approach.

Adaptive middleware monitors and controls 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 [7,23]. 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 [24]. Some of the challenges include the ordering of operations, duplication of operations, recovery, and consistency in the face of multithreading.

A combination of existing dynamic adaptation technologies with a sound engineering approach holds promise to make self-adaptation an engineering reality.

2.2 Prior Self-adaptive Approaches

To date, several dynamic adaptation frameworks have been proposed and developed [8,16,31]. Of these, perhaps the most closely related systems are the architecture evolution framework of Taylor and colleagues from U.C. Irvine and the self-organizing systems of Kramer and colleagues from Imperial College, U.K.

Gorlick and colleagues have developed a framework, Weaves, that 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 [17]. Inspired by the dynamic observation and reconfiguration capability demonstrated in this work, our work broadens support to other styles.

In his dissertation on the "open architecture software" approach, Peyman Oreizy proposed the use of an application's architectural system model as a basis for decentralized software evolution for a greater degree of adaptability while supporting increased assured consistency over previous software evolution techniques [25]. His approach introduced an "architecture evolution manager" to validate changes to the architectural model and to carry out the changes on the application's implementation to reflect the model. Associated with his approach, the ArchStudio environment comprises a number of tools to support evolution of software via changes to the architectural model for C2-style applications. While Oreizy's thesis provided an approach for developers to evolve a system by changing its architectural model at *design time*, our work focuses on enabling monitoring and adaptation of a system consistent with its architectural model at *run time*.

As a natural extension, Oreizy and colleagues added a planning loop to his software evolution approach and introduced an architecture-based run-time software evolution framework [8,26]. As with all architecture-based adaptation, the UCI "architecture evolution framework" dynamically evolves systems using a monitoring and execution loop controlled by a planning loop. This framework, built over the

course of several years, supports self-adaptation for systems built in the C2 hierarchical publish-subscribe style. Evolution of the architectural model uses architectural differencing and merging techniques similar to those used to version-control code. Although powerful and demonstrated on quite a number systems, this approach would be difficult and costly to apply on a target system that deviates from the publish-subscribe style or uses a completely different style. Our work overcomes this limitation by providing a general self-adaptation framework that can be tailored to specific classes of systems.

The work on self-organizing systems proposes an approach where self-managing units coordinate toward a common model, an architectural structure defined using the architectural formalism of Darwin [16]. Each self-organizing component is responsible for managing its own adaptation with respect to the overall system and requires the global architectural model to do so. While this approach provides some advantages 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. Our approach aims to overcome that limitation by allowing tailorable global reorganization without imposing a high performance overhead.

3 Requirements for an Engineering Solution

To improve on the state of current practice and overcome the limitations of the current state-of-the-art, we need an engineering approach that helps software developers achieve external system adaptation in a principled and cost-effective way. In particular such an approach should have three important properties:

- *Generality*. The approach should be applicable to a wide variety of systems and properties. It should not be limited to a specific class of system such as client-server or a single system concern such as performance. For example, a developer should be able to apply the approach with relative ease to a pipe-filter, repository, or event-based system as well as client-server. The developer should also be able to tackle a combination of performance, security, reliability, as well as other prominent run-time system properties using this approach. In addition, the approach should be applicable to both new and existing software-based systems.
- *Cost-effectiveness*. The approach should allow developers to realize and implement self-adaptation capabilities on supported classes of systems at a relatively low-cost compared to development from scratch (perhaps order(s) of magnitude lower), and in a reasonably short amount of time (possibly on the order of a few man-weeks). The approach should not require substantial change to legacy systems. In addition, a self-adaptation solution previously applied to a system should be largely reusable in another system with similar self-adaptation needs.
- *Composability*. The approach should allow self-adaptation capabilities of different domains of concern, e.g., performance, cost, and security, to be specified independently by domain experts. Developers should then be able to

compose these capabilities to achieve self-adaptation for a combination of concerns. The property relies on an expert's ability to analyze the effectiveness of the capabilities he specified. Fortunately, separating the concerns facilitates such analysis. Another implication is that independently specified self-adaptation capabilities would be reusable for similar concerns in different systems.

3.1 Making Self-adaptation External

In practice, most systems deployed today do not satisfy these requirements. Systems that do self-adapt today have application-specific and ``hardwired" self-adaptation capabilities that are difficult to generalize. Such built-in (*internal*) capabilities are often able to detect a problem close to its error source through low-level mechanisms such as exceptions and time-outs. Yet, at the same time, the code is limited to a localized view of the system, making it difficult to detect and correct overall system anomalies such as decreasing end-to-end system throughput. In addition, this internal approach disperses the adaptation logic throughout the system, making it costly and difficult to modify and maintain, hence *not* cost-effective. Embedded and dispersed logic also makes it challenging to reason about the outcome, making composability difficult to achieve. Finally, internal and dispersed logic makes reuse nearly impossible, so developing new self-adaptive systems requires significant duplication of effort and, thus, high cost.

To realize the goal of having general, analyzable, composable, and cost-effective adaptation requires that the adaptation be extracted from actual system code and treated as separate from the system. In fact, a number of recent research efforts use external mechanisms to monitor and adapt a running-system in a closed-loop control fashion [1,26,30]. The closed-loop control paradigm, as illustrated in Fig. 1, provides us leverage to "divide and conquer" the self-adaptation problem, separating the approach into three phases: monitoring, modeling, and control of the target system.

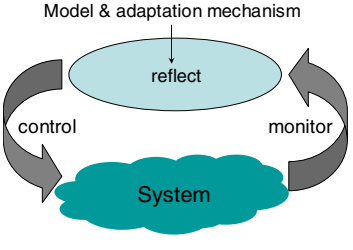

Fig. 1. Illustration of Closed-Loop Control

In principle, external adaptation mechanisms have a number of benefits over internal mechanisms. External control separates the concerns of system functionality from the concerns of "exceptional behaviors," enabling the engineer to systematically focus on and design solutions for dynamic adaptation. As a separate entity, the effectiveness of the adaptation logic is more analyzable and the mechanism more modifiable and extendible. These engineering traits allow the engineer to focus

development, facilitate reuse, and reduce cost. In turn, the engineer can generalize techniques and solutions for adaptation to different and even multiple kinds of systems and system properties. Furthermore, the separation of mechanisms allows this technique to be applied to systems where the source code is not available. This relies on a key assumption that the target system provides, or can be wrapped to provide, hooks to get information out of the system and to make changes.

3.2 Scenarios for Self-adaptation

To clearly address the challenges of self-adaptation, one must understand the scenarios or conditions under which self-adaptation may need to occur. We recognize three major types of conditions for self-adaptation: system errors, changes in the environment of the target system including resource variability, and changes in user preferences. Understanding these different conditions for self-adaptation directly affects the development of capabilities for measuring, modeling, and controlling the target system to support self-adaptation.

A *system error* covers an undesirable condition that arises from the target system itself. For example, a server component may fail, or a set of network connections may go down. An *environment change* and, in particular *resource variability,* covers an undesirable condition that often arises outside the target system and causes problems for the target system. For instance, the wireless network on which an application depends may change beneath it, causing a sudden disruption of connection or change in available bandwidth. Or, the context in which a device is used, such as a room, may change, thus altering the set of resources available to that device. A *change in user priority or preference* constitutes a change in some requirements on the target system. For instance, the user may require high-fidelity video streams at one moment but be satisfied with low fidelity at another.

These three types of conditions share the common property of being a change that may not have been anticipated when assumptions about the intended use of the system were made during system development. These conditions at system run time thus provide opportunities for improvements to bring the system back within the boundaries of its requirements under the newly encountered conditions.

4 Role of Software Architecture

A key issue in applying an external control model is to determine the appropriate kind of models to use as a basis for control decisions. A recent branch of work suggests an architectural model of the software as a useful basis for making decisions about system adaptation [14,26].

4.1 Architectural Model

An *architectural model* represents the system architecture as a graph of interacting computational elements.[1] We adopt a standard view of software architecture that is

[1] Although there are different views of architecture, we are primarily interested in the run-time component-connector view [5].

typically used today at design time to characterize a system to be built. Nodes in the graph, called components, represent the system's principal computational elements and data stores, including clients, servers, databases, and user interfaces. Arcs, called connectors, represent the path-ways for interaction between the components. Additionally, architectural elements may be annotated with various properties, such as expected throughputs, latencies, and protocols of interaction. Components themselves may represent complex systems, represented hierarchically as sub-architectures.

However, unlike traditional uses of software architecture as strictly a design-time artifact, our approach includes a system's architectural model in its run-time infrastructure. In particular, developers of self-adaptation capabilities use a system's software architectural model to monitor and reason about the system. Using a system's architecture as a control model for self-adaptation holds promise in several areas. As an abstract model, an architecture can provide a global perspective of the system and expose important system-level behaviors and properties. As a locus of high-level system design decisions, an architectural model can make a system's topological and behavioral constraints explicit, establishing an envelope of allowed changes and helping to ensure the validity of a change.

Using the architectural model as a basis to monitor and adapt a running system is known as *architecture-based self-adaptation*. A number of researchers have investigated this form of self-adaptation [17,21,26]. Their self-adaptive systems have been hand-crafted to provide strong support for particular classes of system (e.g., data-flow) and to target specific domains of concern (e.g., performance). Given a system in a supported system class, there will typically be an architecture description language and tool support to analyze and model the system, capture constraints on system behavior, detect constraint violations, and adapt the system.

4.2 Architectural Style

To capture system commonalities, we adapt the notion of an architectural style. Traditionally, the software engineering community has used architectural styles to help encode and express system-specific knowledge [29]. An architectural style characterizes a family of systems related by shared structural and semantic properties. The style is typically defined by four sets of entities:

- Component and connector types provide a vocabulary of elements, including components such as Database, Client, Server, and Filter; connectors such as SQL, HTTP, RPC, and Pipe; and component and connector interfaces.
- Constraints determine the permitted composition of the elements instantiated from the types. For example, constraints might prohibit cycles in a particular pipe-filter style, or define a compositional pattern such as the starfish arrangement of a blackboard system or a compiler's pipelined decomposition.
- Properties are attributes of the component and connector types, and provide analytic, behavioral, or semantic information. For example, load and service time properties might be characteristic of servers in a performance-specific client-server style, while transfer-rate might be a property in a pipe-filter style.
- Analyses can be performed on systems built in an appropriate architectural style. Examples include performance analysis using queuing theory in a client-server system, and schedulability analysis for a real-time-oriented style.

To support the needs of run-time system self-adaptation, we augment the notion of style with the notions of operators (to change an architecture) and adaptation strategies (to package changes for specific purpose). In previous work, we have extensively described the significant leverage that architectural style affords us [3]. That is, style provides opportunity for specific analysis of system behavior and properties. For self-adaptation, each style may uniquely guide the choice of metrics, help identify strategic points for system observation, and suggest possible adaptations.

5 The Rainbow Framework

In this section, we briefly introduce the Rainbow framework, which has already been reported in prior work [3,13]. In this paper, we focus on the separation between the general parts of Rainbow that can be applied to a wide variety of systems, and the tailorable parts that need to be written to apply Rainbow to specific systems and concerns.

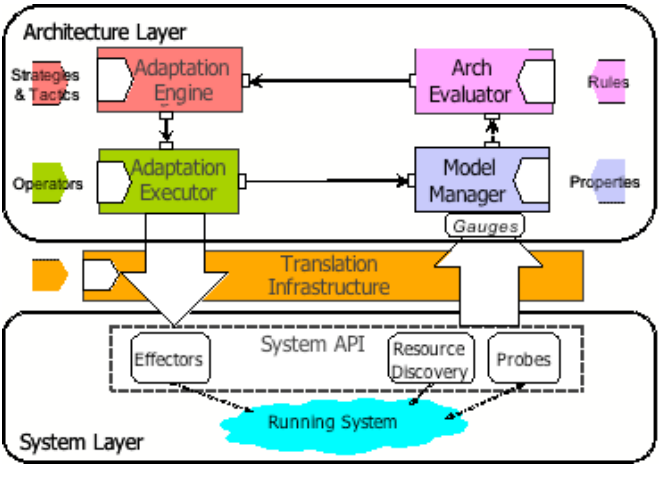

Fig. 2. The two-part Rainbow Framework

Rainbow uses the architectural model of the system to monitor the system and "reason" about appropriate actions. Monitoring mechanisms observe the running system. Observations are related to properties of the architectural model via *probes* and *gauges* [1,15]. The model is periodically evaluated to ensure that the system is operating within an envelope of acceptable range. If the evaluation determines that the system is not operating within the accepted range, the adaptation process is triggered, which determines the action to take. The adaptation is executed on the running system via system-level effectors.

The key idea for applying Rainbow in different situations is the separation of the framework into two parts (see Fig. 2). The first comprises a set of general, common infrastructures that are reusable across systems. The second consists of tailorable parts that can be specialized and customized for particular styles of system and various

system properties of concern. The reusable infrastructures consist of the monitoring mechanisms, the model manager, the architectural evaluator, the adaptation engine, the executor, and various translators. The tailorable parts determine what properties of the system to monitor, what rules or constraints to evaluate, what adaptation actions to take when constraints are violated, and how to carry out those adaptations in terms of architectural as well as system-level operators.

This two-part self-adaptation approach has a number of advantages. By providing a substantial base of reusable infrastructure it greatly reduces the cost of development. By providing separate tailoring parts it 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 domains of concern, and to add and evolve adaptation capabilities with ease. Furthermore, as described later, a modular adaptation language for tailoring the adaptation mechanism allows engineers to consider adaptation concerns separately and then put them to work together. In short, assessed abstractly, the Rainbow approach has the potential to satisfy the generality, cost-effectiveness, and composability requirements set forth initially.

6 Case Instantiations of Rainbow

To date, the Rainbow framework has been instantiated in two case study systems, and a third case study is in progress. Each case study has demonstrated the application of Rainbow to a different style of system, a different kind of system concern, as well as slightly different subsets of Rainbow capabilities.

The first case study provided an end-to-end investigation of the feasibility of the architecture style-based self-adaptation approach, and demonstrated effectiveness through an experiment on a dedicated testbed consisting of five routers and 11 machines communicating over 10-megabits-per-second lines [13]. The second case study demonstrated the potential generality of Rainbow applied to a different architectural style and an additional property of concern over the first case study, as well as revealed a moderate framework computational overhead [4]. Moreover, these two case studies helped to distill the reusable infrastructures of the framework [12]. The third case study in progress aims to show generality with a third data point on the kinds of system concern that Rainbow can address.

Here, we focus on how the two-part framework is instantiated for each of the case-studies to show how it can make self-adaptation an engineering reality.

6.1 Case 1: Client-Server Style with Performance Concern

In the first case study, we experimented with the application of Rainbow to a client-and-server style system, which consisted of a number of clients connected to one cluster of servers, with a specific performance concern of latency. The results show that for this application and the specific loads used in the experiment, self-adaptation significantly improved system performance. Fig. 3 shows sample results for system performance with and without adaptation. Fig. 3a shows that, without adaptation, once the latency experienced by each client rises above 2 seconds, it never again falls

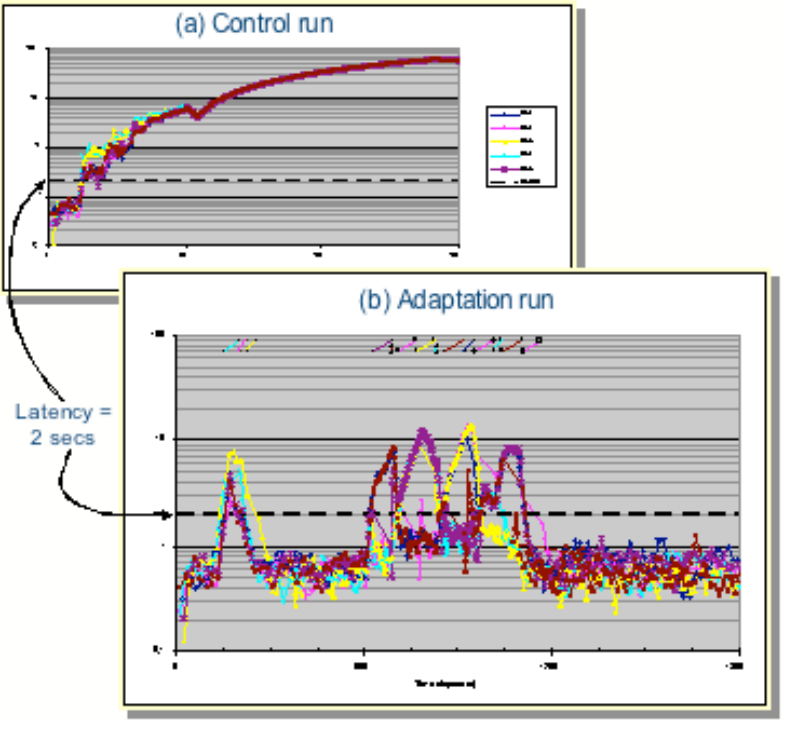

Fig. 3. System performance with and without self-adaptation. The dashed lines indicate the desired latency behavior

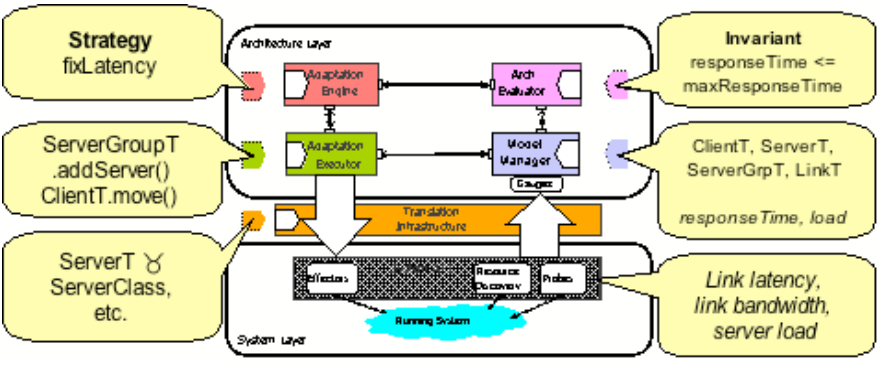

Fig. 4. Rainbow instantiation for client-server and performance

below this threshold. On the other hand, Fig. 3b shows that if Rainbow issues the adaptations, the client latencies return to optimal levels.

For this case study, as illustrated in Fig. 4, a vocabulary for the client-server style system elements is defined, along with performance-related properties. Specific performance properties—latency, bandwidth, load—are identified for monitoring. An

invariant is defined over the system's architectural model to indicate the condition for adaptation. Thus, the architectural evaluator "sounds an alarm" when the response time of a client rises above some maximum threshold. A strategy has been defined to deal with this latency issue, and the strategy uses style-specific architectural operators such as *addServer*() and *move*(). A mapping helps to translate elements and actions in the architectural level to their counterparts in the system level. Note that, in general, mappings between architecture-layer and system-layer elements and actions may not be one-to-one, but will often be one-to-many.

6.2 Case 2: Service Coalition Style with Performance and Cost Concerns

In the second case study, we investigated the use of Rainbow on a service-coalition style, video-conferencing system with a simultaneous need to provide good-quality video service while keeping cost down to the customers using the service. Perhaps not surprisingly, this case study revealed that self-adaptation incurs some latency. In this system, the lapsed time for adaptation at the architecture, translation, and system layers were 230, 300, and 1,600 ms, respectively, for one scenario, and 330, 900, and 1,500 ms, respectively, for another. These results indicate that the software architecture-based approach best suits adaptations that operate on a system-wide scale and fix longer-term system behavior trends.

Because this system shared common performance properties with the first case study, we were able to reuse parts of the monitor and control infrastructures. In fact, from the first case study, the Rainbow prototype reused approximately 100 kilo-lines of code out of 102, plus an additional 73 kilo-lines of tool and utility code.

For this case study, as shown in Fig. 5, a vocabulary for the service coalition style system elements is defined, along with performance and cost-related properties. Specific properties—cost, load, bandwidth—are identified for monitoring. A few invariants are defined over the system's architectural model to indicate the conditions for adaptation. In particular, the architectural evaluator "sounds an alarm" either when the available bandwidth on certain connections drop below a minimum threshold, or the cost of serving the users rise above a maximum threshold. Adaptation strategies have been defined to deal with these two kinds of issues, and the strategies use style-specific architectural move() operators.

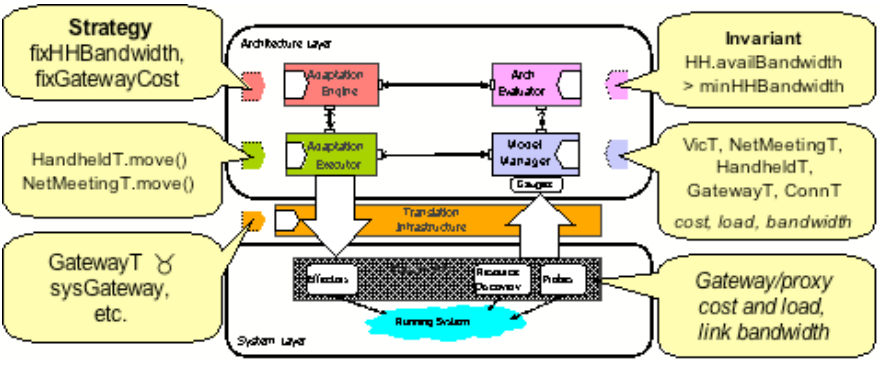

Fig. 5. Rainbow instantiation for service coalition and performance + cost

6.3 Case 3: Client-Server Style with Security Concern

The third case study is a straightforward client-server style system where a set of servers processes the requests of many clients, some of which can be malicious. The primary concern in this system is security, specifically, to appropriately detect and adapt against distributed denial of service attacks without greatly compromising such attributes as data integrity. Again, due to certain commonalities between this system and the previous two case studies, a significant subset of the framework, in addition to the common infrastructures, will be reusable.

In this case study, as illustrated in Fig. 6, a vocabulary for the specific kind of client-server style system elements is defined, along with security-related properties. Specific security properties—load, intrusion patterns—are identified for monitoring. An invariant is defined to trigger adaptation when the intrusion probability rises above a maximum threshold. Multiple adaptation strategies have been defined to deal with intrusion, including partitioning the network and securing data. The strategies use style-specific architectural operators supported by the various system elements.

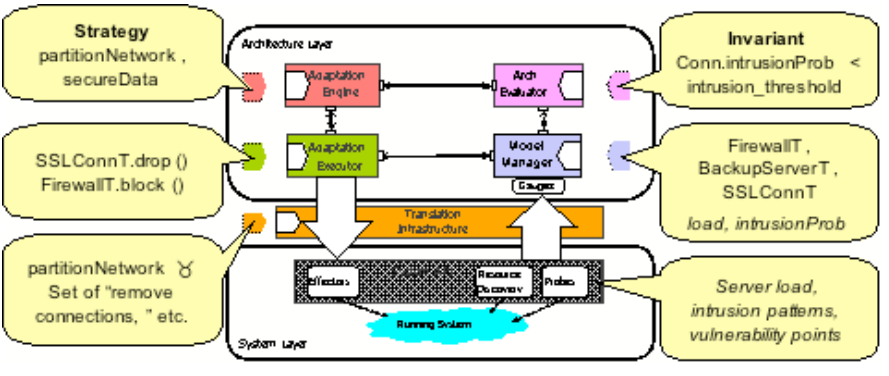

Fig. 6. Rainbow instantiation for client-server and security

6.4 Generalizing Across Cases

Table 1 summarizes the three case studies along two dimensions—architectural style and system concern. We believe that the Rainbow framework applies to a sufficient number of data points in these two dimensions to demonstrate generality.

Table 1. Summary of Rainbow case studies to date

Concern \ Style	Client-Server	Service-Coalition	…	Repository	…
Performance	X	X			
Cost		X			
Security	X				
….					
Reliability					
…					

These three case studies illustrate how we envision the application of Rainbow and give concrete examples to help characterize the tailorable parts. These case studies also bring into focus the aspects of the Rainbow framework that are (or need to be) reusable and generalized across systems. As a result, we have identified a number of key challenges to make the Rainbow approach possible.

In particular, we are presently investigating two research challenges that arise from the ability in Rainbow to tailor specific adaptation strategies for different styles of systems and different kinds of system concern, namely representing adaptation knowledge and coordinating adaptations. A general ability to represent adaptation knowledge allows engineers to "plug-in" different strategies, making adaptation external and modifiable. This marks an important step toward providing *cost-effective* self-adaptation. A general mechanism to resolve conflicts and coordinate adaptations allows system engineers to consider system properties of concern separately and *compose* adaptation strategies.

As these challenges are being resolved, the Rainbow framework holds promise to realize our vision and make self-adaptation an engineering reality.

7 Looking Forward

In this paper, we have expressed our vision of a software engineering reality where engineers can develop self-adaptive software-intensive systems cost-effectively. We have discussed the Rainbow approach and shown how it charts a path to realizing this vision. Specifically, we have described three case studies of systems with different concerns and qualitatively demonstrated how Rainbow generalizes across different styles of systems and different concerns.

In a larger context, the Rainbow framework holds potential for application in other forms of composition, particularly in relation to human task, autonomic computing, and software design. Recall that one of the conditions of self-adaptation is changes in user priority or preference. For Rainbow to be aware of such changes, it would provide appropriate interfaces to the user level. Specifically, a user would be able to influence the behavior of the framework via such variables as frequency and granularity of monitoring, choice of adaptations, and quality of actions taken.

Recently, there has been a push by IBM and others toward developing systems or elements of systems that are autonomic. That is, they are self-managing and exhibit self-configuring, self-healing, self-protecting, and self-optimizing properties [11]. Since Rainbow provides the mechanisms for self-adaptive systems, it is possible to apply Rainbow in the context of autonomic systems to construct a system of systems in which each constituent system exhibits self-adaptive capabilities.

Finally, certain insights from the Rainbow approach can influence how software engineers design future software. One of the main assumptions of the Rainbow approach is that we require the target system to provide hooks for measuring and changing the system. What if the software is designed to provide such hooks? Indeed,

if the software engineering community standardizes interfaces for extracting information from and effecting changes on systems, engineers would be able to produce systems that plug-and-play with self-adaptation infrastructures like Rainbow.

Acknowledgements

This research was supported by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, by the US Army Research Office (ARO) under grant numbers DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab and DAAD19-01-1-0485, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. The views and conclusions described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, NASA, the US government, or any other entity.

References

1. Robert Balzer. Probe run-time infrastructure. http://schafercorpballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt, 2001.
2. 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.
3. 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 (WICSA-3), pages 45–59, Massachusetts, USA, August 25–30, 2002. Kluwer Academic Publishers.
4. Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. An architecture for coordinating multiple self-management systems. In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4), Oslo, Norway, June 11–14, 2004.
5. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford, editors. Documenting Software Architecture: Views and Beyond. The SEI Series in Software Engineering. Pearson Education, Inc., 2003.
6. Nathan Combs and Jeff Vagel. Adaptive mirroring of system of systems architectures. In Garlan et al. [14], pages 96–98.
7. Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In USENIX Summer, pages 267–278, 1994.
8. Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. [14], pages 21–26.
9. Jason Flinn, SoYoung Park, and Mahadev Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), pages 217–226. IEEE Computer Society Press, July 02–05, 2002.

10. Colleen Frye. Self-healing systems. Application Development Trends, pages 29–34, September 2003.
11. A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. IBM Systems Journal, 42(1):5–18, 2003.
12. 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):46–54, October 2004.
13. 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.
14. David Garlan, Jeff Kramer, and Alexander Wolf, editors. Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), New York, NY, USA, November 18–19, 2002. ACM Press.
15. David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In [28].
16. Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organizing software architectures for distributed systems. In Garlan et al. [14], pages 33–38.
17. Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis. In 13th International Conference of Software Engineering, pages 23–34, Los Alamitos, CA, USA, May 1991. IEEE, IEEE Computer Society Press.
18. Phil Gross. MEET. http://www.psl.cs.columbia.edu/meet/index.html, 2002.
19. Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S. Kc, and Janak J. Parekh. An active events model for systems monitoring. In [28].
20. Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. In Communications of the ACM, 45(6), pages 33–38, June 2002.
21. Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. InW. 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.
22. V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. IBM Systems Journal, 42(1):98–106, 2003.
23. Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. IEEE Computer, pages 62–68, July 1999.
24. 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.
25. Peyman Oreizy. Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. PhD thesis, University of California, Irvine, 2000.
26. 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.
27. Proceedings of the International Conference on Autonomic Computing, New York, NY, May 17-18, 2004.
28. Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 12–14, 2001.

29. Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
30. Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In Garlan et al. [14], pages 73–78.
31. 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 [28].