Submitted for publication *19 January 2004*

Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure

Shang-Wen Cheng An-Cheng Huang David Garlan Bradley Schmerl Peter Steenkiste
School of Computer Science, Carnegie Mellon University
{zensoul, pach, garlan, schmerl, prs}+@cs.cmu.edu

Abstract

Software-based systems today are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. Recent work uses closed-loop control based on external models to monitor and adapt system behavior at run time. Taking this externalized approach, the Rainbow framework we have developed uses software architectural models to dynamically monitor and adapt a running system. A key goal of this framework, and also a primary challenge, is to support the reuse of adaptation strategies and infrastructure across different systems. In this paper, we show using two case studies that the separation of a generic adaptation infrastructure from system-specific adaptation knowledge enables Rainbow to be applied to different styles of systems.

1 Introduction and Motivation

Software-based systems today increasingly operate in changing environments with variable user needs, resulting in a continued rise of administrative overhead for managing these systems. Thus, systems are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. Although mechanisms that support self-adaptation exist in the form of programming language features and algorithms, most adaptation mechanisms found in existing systems are highly specific to the application and tightly bound to the code. This "internalized" approach is brittle, costly to change, and difficult to reuse in a new system. In contrast, recent work uses a closed-loop control based on external models and mechanisms to monitor and adapt system behavior at run time to achieve various goals [8, 15, 14, 20, 24].

In principle, the strength of externalized control mechanisms over "internalized mechanisms" lies in separating the concerns of functionality from the concerns of "exceptional behaviors." This separation of concerns allows the effects of adaptation to be analyzed. The adaptation mechanisms can be modularized to facilitate modification and extension. Furthermore, separating adaptation mechanisms from an application enables this approach to be applied to legacy systems where the source code may not be available. Finally, modularized adaptation mechanisms enable reuse and potentially reduce the cost of adding self-adaptation.

This externalized approach requires an appropriate model of the system to allow reasoning about the system's behavior. Several researchers have proposed the use of architectural models, which represent the system as a gross composition of components and their interconnections. This approach is known as architecture-based self-adaptation [8, 14, 24]. Among the many benefits, an abstract architectural model can provide a global perspective of the system and expose important system-level properties and integrity constraints.

In our research, we have developed a framework, Rainbow, which uses architectural models to dynamically monitor and adapt a running system, often at a fairly global, module level. In previous work ([3]), we have demonstrated the feasibility of Rainbow in dynamically adapting target systems. One of the promises of Rainbow is to provide a low-cost approach for adding self-adaptation capabilities to a wide range of systems. Achieving this generality is the primary challenge we address in this paper. Specifically, we will show the extent of sharing or "reuse" of the framework across different systems.

A key idea to enable reuse is to leverage the notion of *architectural style*. Style specifies a vocabulary of component and connector types and the rules of system composition, essentially capturing the units of change from system to system. To support self-adaptation, different styles will typically provide different metrics and strategic points for observing, different techniques for reasoning about, and different capabilities for adapting a system.

In this paper, we present the Rainbow approach and describe how we use software architectural style to separate system-specific adaptation knowledge from the adaptation infrastructure, thereby allowing principled reuse. We then present two case studies that demonstrate the reuse of infrastructure across two systems with different styles.

2 The Rainbow Approach

The primary objective in designing the Rainbow framework is to provide a principled, time-saving, and low-cost solution for software developers to add self-adaptation capabilities to a wide range of systems. Figure 1 shows the control loop of the Rainbow framework for self-adaptation. Rainbow monitors the run-time properties of an executing system in the system-layer through an abstract model maintained by the model manager in the architecture layer. The constraint evaluator checks the model for constraint violation and triggers the adaptation engine if a problem occurs. The adaptation engine and the executor then perform adaptations on the running system. The architecture and system layers interact through a translation layer.

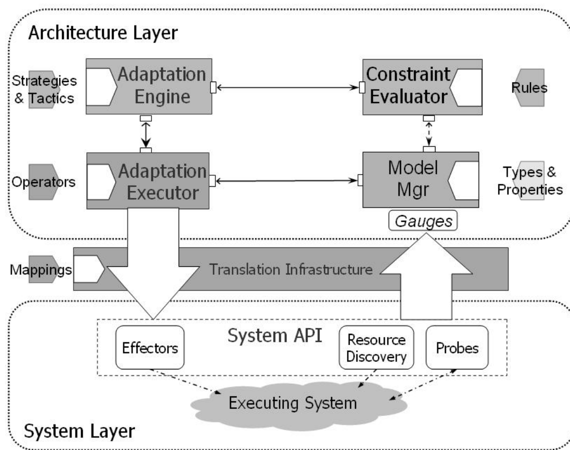

Figure 1. The Rainbow framework.

2.1 Software architectures

In the Rainbow framework the architectural model of the system is used to monitor and reason about the system. The architecture of a software system is an abstract view of the system as a composition of computational elements and their interconnections [26]. An *architectural model* represents the system architecture as a graph of interacting components, as illustrated in Figure 2.[1] This is the core architectural representation scheme adopted by a number of architecture description languages (ADLs), including Acme [12], xADL [7], and SADL [22]. Nodes in the graph are termed *components*. They represent the principal computational elements and data stores of the system: clients, servers, databases, user interfaces, etc. Arcs are termed *connectors*, and represent the pathways of interaction between the components. A given connector may in general be realized in a running system by a complex base of middleware and distributed systems support.

[1] Although there are different views of architecture [6], we are primarily interested in the component-connector view.

The use of a system's architecture as a control model for self-adaptation holds a number of potential promises. As an abstract model, an architecture can provide a global perspective on the system and expose the 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, thereby not only establishing an envelope of allowed changes, but also helping to ensure the validity of a change.

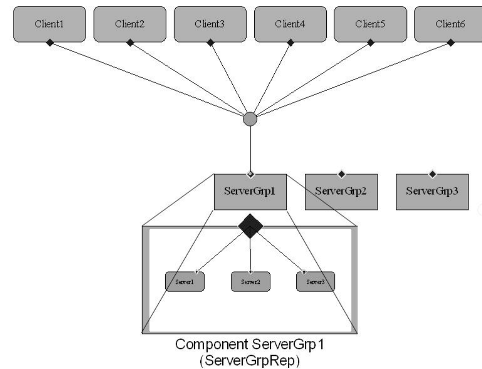

Figure 2. Software architecture of a client-server system.

Consider, for example, a web-based client-server system with a group of clients requesting files from a set of server farms. The architecture of the system, shown in Figure 2, is a client-server model that captures six Client components and three ServerGroup components, each of which encapsulates a number of Server components. In the figure, we've shown the expansion of the first ServerGroup. The Clients and a ServerGroup are joined by a communication connector. This architectural model provides a global perspective on the system by revealing all the components and how they are connected. The elements in the model contain such important properties as the load of each Server, the bandwidth of each connection, and the response-time that each Client experiences. Furthermore, the model maintains explicit constraints on the architecture that, for instance, requires each Client to connect to exactly one ServerGroup. The constraints establish an envelope of allowed changes by ensuring, for example, that no future changes to the system leaves a Client dangling without connection. A change to the system is valid if the system satisfies the constraints after the change.

2.2 Reusable units of Rainbow

To fulfill Rainbow's objective, the various components of the framework need to be reusable from system to system. To identify what parts of the framework can be reusable, and under what circumstances, we divide the framework into an *adaptation infrastructure* and the *system-specific adap-*

tation knowledge (see Figure 1), described next. The adaptation infrastructure, divided into system, architecture, and translation layers, provides common functionalities across self-adapting systems and are therefore reusable, while the adaptation knowledge may be more difficult to reuse.

System-layer infrastructure. At the system layer, we have defined the necessary system access interface and built an infrastructure that implements the interface. A system measurement mechanism, realized as probes, observes and measures various states of the system. The system information may be published by or queried from the probes. A resource discovery mechanism can be queried for new resources based on resource type and other criteria. An effector mechanism carries out the actual system modification.

Architecture-layer infrastructure. At the architecture layer, *gauges* aggregate information from the probes and update the appropriate properties in the model. A model manager manages and provides access to the architectural model of the system. A constraint evaluator checks the model periodically and triggers adaptation if a constraint violation occurs. An adaptation engine determines the course of action and carries out the necessary adaptation.

Translation infrastructure. A translation infrastructure helps to mediate the mapping of information across the abstraction gap from the system to the model and vice versa. A translation repository within the infrastructure maintains various mappings shared by the translator components to, for example, translate an architectural-level element identifier into an IP address or an architectural-level change operator into some system-level operations.

System-specific adaptation knowledge. To add self-adaptation to a system, the adaptation infrastructure needs to be tailored using the system-specific adaptation knowledge, which includes the types and properties of components, behavioral constraints, and strategies for adaptation.

Ideally, all of the adaptation infrastructure should be reusable across systems. However, the generality of the framework comes at a cost: a significant amount of system-specific knowledge needs to be specified to apply the framework to different systems. To address this, we show that the notion of *architectural style*, discussed next, will allow reuse of a large portion of system-specific adaptation knowledge between systems in similar domains. Moreover, the supporting architectural description language and development-time toolsets for defining the architectural models and tailoring self-adaptation should also be completely reusable.

2.3 Architectural style

Traditionally in the software engineering community, *architectural styles* have been used at design time to help encode and express system-specific knowledge in the system architectures. An architectural style characterizes a family of systems related by shared structural and semantic properties, and typically provides four sets of things [2, 21]:

Types. A These define a vocabulary of element types, including component types such as Database, Client, Server, Filter; connector types such as Sql, Http, Rpc, Pipe; and component and connector interface types.

Rules. These are constraints that determine the permitted composition of the elements instantiated from the types. For example, the rules might 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. Rules may also take on the more lenient form of heuristics (e.g., to guide system adaptation).

Properties. These are attributes characteristic of element *types* in a style to provide analytic and sometimes behavioral or semantic information. For instance, *load* and *service-time* properties might be characteristic of Servers in a performance-specific client-server style, while *transfer-rate* might be a common property in a Pipe of a pipe-filter style.

Analyses. Analyses make use of *properties* and are performed on systems built in that style. Examples include performance analysis using queueing theory for a client-server system [27] and schedulability analysis for a real-time oriented style.

These four sets capture the static attributes of the system. Supporting run-time adaptation also requires capturing the dynamic attributes of the system, both in terms of what operations can be done on the system, as well as how the operations can be applied. Therefore, we need to augment the notion of architectural style with operators and strategies for adaptation, resulting in an extended notion we call an *adaptation style*.

Operators. These include adaptation operators defined for particular element *types*, and query operators defined for element *properties*. Adaptation operators determine a set of style-specific actions 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. Query operators allow values of system properties to be retrieved on demand.

Strategies. A strategy, defined in terms of *operators* and *properties*, prescribes how to adapt a system in response to an undesirable condition. For instance, a service-coalition system may have a system-wide cost constraint. Upon violating it, an adaptation strategy might progressively replace the most costly service with lower-grade service until the overall cost is once again within bounds.

Although strategies use operators and properties of an adaptation style to adapt systems in that style, they are designed for particular *system concerns*. A system concern refers to a related set of system requirements (e.g., performance, cost, or reliability) and determines the set of system properties on which self-adaptation should be focused, and hence the set of strategies. For a system, the properties determined by the system concern is a subset of the properties in that system's style. For example, a client-server system may have a style that includes load, bandwidth, and cost properties while a particular performance concern on the system focuses only on the load and bandwidth properties.

Adaptation style and system concern together comprise two important dimensions of variability from system to system. Reuse of the system-specific adaptation knowledge in the framework will depend on how similar two systems are in terms of their styles and their system concerns. More specifically, types, properties, adaptation strategies, and architectural operators may be reusable if the style and concerns of the two systems match.

3 Adaptation Reuse

In this section, we present two case study systems that have different adaptation styles but share the same system concern. We show how the various units of the Rainbow framework get reused across the two prototyped systems. We also describe the implementation of the adaptation infrastructure and present some preliminary evaluation results.

3.1 Web-based client-server system

The first case study system (see Figure 2) consists of a set of web clients, each of which makes stateless requests of contents from one of several groups of web servers. The client and server components are implemented in Java and provide RMI interfaces for the effectors to perform adaptation operations. Clients connected to a server group send requests to the shared request queue of the group, and servers that belong to the group grab requests from the queue.

The primary system concern is performance, or, more specifically, the response time experienced by the clients. An analysis of the system using queueing theory identifies that the server load and available bandwidth are two properties that affect the response time [4].

Based on the system concern and the analysis above, a client-server style is defined for the system. The major parts of the style are presented below (for simplicity, not everything is shown, e.g., some operators are omitted).

- Types: ClientT, ServerT, ServerGroupT, LinkT.
- Properties: ClientT.responseTime, ServerT.load, ServerGroupT.load, LinkT.bandwidth.
- Operators:
  - ServerGroupT.addServer(): finds and adds an available ServerT to a ServerGroupT to increase the capacity.
  - ClientT.move(ServerGroupT toGroup): disconnects ClientT from its current ServerGroupT, then connects ClientT to the *toGroup* ServerGroupT.

One example adaptation strategy based on the concern and analysis is as follows:

```
invariant C : responseTime <= maxResponseTime
! → responseTimeStrategy(C);

strategy responseTimeStrategy (C : ClientT) {
    let G : ServerGroupT = findConnectedServerGroup(C);
    if (query("load", G) > maxServerLoad) {
        G.addServer();
        return true;
    }
    let conn : LinkT = findConnector(C, G);
    if (query("bandwidth", conn) < minBandwidth) {
        let G : ServerGroupT = findBestServerGroup(C);
        C.move(G);
        return true;
    }
    return false;
}
```

This adaptation strategy is used by the adaptation engine at run time to maintain the invariant that each client's perceived response time is less than a pre-defined maximum response time. If the invariant is violated, the adaptation engine executes the strategy as follows. First, if the load of the current server group exceeds a pre-defined threshold, a server is added to the group to decrease the load and therefore decrease the response time. Otherwise, if the available bandwidth between the client and the current server group is too low, the client is moved to another group, resulting in higher available bandwidth (and lower response time).

3.2 Video-conferencing system

The second example system is a video-conferencing session that has a set of participating users (Figure 3). Some participants use the Vic/SDR video conferencing tools, which use IP multicast. Some participants use NetMeeting, which speaks the H.323 protocol and only uses unicast. Other participants use a handheld device, which runs a slightly modified version of Vic. In addition, since a handheld device does not have the capability to perform protocol negotiation, a handheld proxy (HHP) is needed to join the conferencing session on behalf of each handheld user. A video conferencing gateway (VGW) [17] that supports both H.323 and the Session Initiation Protocol is used to translate the protocols for NetMeeting and Vic users, Finally, to allow efficient communication among all participants across wide-area networks, the system uses an end-system multicast (ESM) overlay (specifically, Narada [5]) consisting of a number of ESM proxies to provide the multicast functionality.

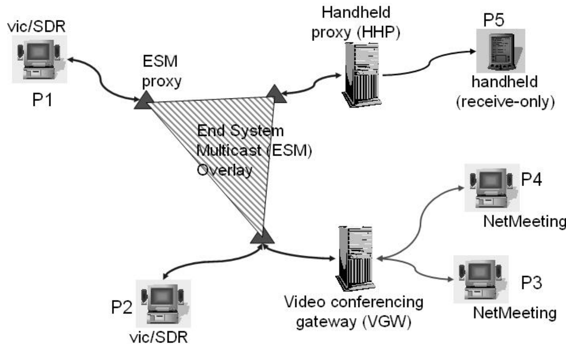

Figure 3. Video-conferencing system

The system concern here includes both performance and cost. For example, one goal is to maintain sufficient available bandwidth between the handheld user and the handheld proxy. Another example is that through analysis, the gateway is identified as the main contributor to the total cost. Therefore, another goal is to keep the unit cost of the gateway low (e.g., if there is only one remaining NetMeeting user, switch to a low-cost gateway).

Based on the system concern and analysis above, a video-conferencing style is defined for the system. The major parts of the style are presented below (for simplicity, not everything is shown).

- Types: VicT, NetMeetingT, HandheldT, GatewayT, HandheldProxyT, ESMProxyT, ConnectionT
- Properties: GatewayT.cost, GatewayT.load, ConnectionT.bandwidth
- Operators:
  - HandheldT.move(HandheldProxyT toHHP): switches handheld user to a new handheld proxy.
  - NetMeetingT.move(GatewayT toVGW): switches NetMeeting user to a new video gateway.

Two example adaptation strategies (expressed in pseudocode) based on the concern and analysis are:

```
invariant HH : available bandwidth between handheld user HH
            and its handheld proxy HHP > minHHBandwidth
! → HHBandwidthStrategy(HH);

invariant VGW : (VGW.cost / number of NM users)
            <= maxVGWUnitCost
! → VGWCostStrategy(VGW);

strategy HHBandwidthStrategy (HH : HandheldT) {
    let HHP1 : HandheldProxyT = findBestHHP(HH);
    HH.move(HHP1);
    return true;
}

strategy VGWCostStrategy (VGW : GatewayT) {
    let VGW1 : GatewayT = the gateway with the lowest cost
                that can handle the current load;
    if ((query("cost", VGW1) / number of NM users)
            < maxVGWUnitCost) {
        for each NetMeeting user U of VGW {
            U.move(VGW1);
        }
        return true;
    }
    return false;
}
```

At run time, when an invariant is violated, the corresponding adaptation strategy is executed by the adaptation engine. For example, when the available bandwidth between the handheld user and the handheld proxy becomes too low, the handheld user is moved to a better handheld proxy. When the unit cost of the gateway VGW becomes too high (e.g., when a NetMeeting user leaves the session), the NetMeeting users connected to VGW are switched to the lowest-cost gateway that can handle the load.

3.3 Discussion of reuse

Now we discuss how the three layers of the adaptation infrastructure and the various parts of style (representing the system-specific adaptation knowledge) are reused across the two case study systems.

3.3.1 System-layer infrastructure

Effectors. Effectors are component-specific, i.e., an effector is capable of performing adaptation operations on a particular type of component. For example, an effector for the video conferencing gateway knows how to activate and shutdown a gateway. Our system-layer infrastructure provides a reusable interface for accessing the effectors, for example, the adaptation engine/executor can issue an "activate" operation (e.g., on either a gateway or a web server) by invoking the corresponding function provided by the system API. The system-layer infrastructure then dispatches the operation to the appropriate effector (e.g., a gateway effector or a web server effector) based on the type of the target component. Since the two case study systems do not share any component types, their effectors cannot be reused.

Probes. As mentioned earlier, probes measure system properties and provide the information to the model manager through push-based (monitoring) or pull-based (querying) methods. In order for the constraint evaluator to check whether the invariants remain true, probes periodically update the relevant properties in the model, for example, each client's perceived response time or available bandwidth for the handheld user. In addition, adaptation strategies often need to query some additional information, for example, the server group load and the cost of the new gateway. Therefore, for our case study systems the system-layer infrastructure provides probes that measure the following properties of various system components: response time, load, and bandwidth. Among these, the load and bandwidth probes are reused across the two systems since these two proper-

ties are of interest in both systems.

Resource discovery. A resource discovery mechanism is needed by the adaptation engine to, for example, find available components to replace existing ones as directed by the adaptation strategy. For instance, for the video conferencing system, the first strategy finds a handheld proxy component that has the most available bandwidth for the handheld user. The second strategy finds a video conferencing gateway component that supports both Vic users and NetMeeting users and has the lowest cost. Our system-layer infrastructure supports resource discovery based on component type and other desired component attributes.

3.3.2 Architecture-layer infrastructure

At the architecture layer of the framework, the adaptation style specifies the types, properties, and rules of the model instance that is managed by the model manager. The model manager, constraint evaluator, and adaptation engine are reused. Gauges for the shared properties of load and bandwidth are reused to aggregate information from the corresponding probes and update the appropriate properties in the model.

3.3.3 Translation infrastructure

The translation layer is needed to bridge the abstraction gap between the model and the system. For example, when the adaptation engine performs resource discovery to find a video conferencing gateway, the translation layer must map the architectural type “GatewayT” to the system-level component type “sysGateway” that is used by the system-layer resource discovery mechanism. Another example is that when the adaptation engine applies the connect operator to two elements (e.g., a NetMeeting user and a gateway) in the architectural model, the translation layer must map them (e.g., represented by element identifiers in the model) to actual machines (e.g., represented by IP addresses and port numbers) in the system. As mentioned earlier, the translation layer handles four kinds of mappings. These mappings are stored in an entity called *translation repository*, and the actual translation is performed by the translators. Although the mappings are system-specific, the repository and translators are reused.

3.3.4 System-specific adaptation knowledge

The two case study systems differ in adaptation styles, but share the same system concern, i.e., the adaptation is concerned with maintaining performance. The concern manifests itself in the properties of each style, so sharing the same concern means that we can reuse the knowledge about the shared properties (load and bandwidth). Part of the knowledge is the translation mappings used to map system properties to architectural properties defined by the style. For properties that need to be aggregated by the gauges (e.g., average latency), the aggregation knowledge (e.g., algorithm) can also be reused.

From the two case studies, we conjecture a more general way of determining what system-specific adaptation knowledge in the framework can be reused in terms of the two dimensions of adaptation style and system concerns, summarized in Table 1.

Table 1. Summary of framework reuse

	Style	Concern	Reuse
1	different	different	Adaptation infrastructure
2	different	same	Adaptation infrastructure, Properties, Mappings
3	same	different	Adaptation infrastructure, Types, Rules, Mappings, Adaptation operators
4	same	same	Adaptation infrastructure, Types, Rules, Properties, Mappings, Adaptation operators

3.4 Implementation and Evaluation

In addition to the two case study systems above, we have implemented a prototype of the Rainbow self-adaptation framework. At the system level, we use the global network positioning (GNP) approach [23] for estimating network latency, the Remos tool [9] for measuring bandwidth, and the network-sensitive service discovery (NSSD) mechanism [18] for resource discovery. We implemented probes that obtained information from GNP and the Remos tool.

The architecture-layer entities are implemented in Java based on the set of Acme architectural design tools developed previously [25]. Performance-property gauges were implemented to read values from the probes and update the model manager’s model via two event broadcast buses [13].

For the translation infrastructure, the translation repository provides a Java RMI interface for the translators to store and retrieve the mappings needed for translation. Some translators are standalone entities, while others are integrated modules of system-layer or architecture-layer entities. Communications within the framework use XML messages over Java RMI.

As an approximation, we use code size as one measure of reuse to evaluate the Rainbow approach. The size of the Rainbow prototype—including the adaptation mechanism, model manager, gauge and probes and their infrastructure, translation infrastructure, and system-layer infrastructure that we implemented—is 105 kilo-lines of code (KLoCs), with a breakdown of 86, 12, and 7 KLoCs for the architecture, system, and translation layers, respectively. Of these, the reused code for the adaptation infrastructure is about

103 KLoCs (98.2%). In addition, 73 KLoCs of tool and utility were reused.

Two other important aspects to evaluate are the effectiveness of self-adaptation and the performance of the Rainbow framework. We have already reported on the effectiveness of self-adaptation in previous work using the client-server system [4]. The results demonstrate that when the response time for a client exceeds the pre-defined threshold, the framework is able to detect the problem, perform the appropriate adaptation (adding a server or switching server group), and bring the response time below or near the threshold. We then use the video conferencing system to measure the performance of the framework in two adaptation scenarios. The elapsed time for adaptation at the architecture, translation, and system layers are, respectively, 230, 300, and 1600 ms for one scenario, and 330, 900, and 1500 ms for another scenario. We believe the performance is reasonable, for the inevitable overhead incurred at the system layer is higher than the other two layers combined.

4 Discussion

The Rainbow approach rests on a couple of important assumptions. One assumption is that for any target system, the framework has access to some measurement, resource discovery, and effecting mechanisms to observe and change the system. We believe this assumption is reasonable because a growing number of measurement tools and infrastructures are able to provide measurement or information about common component properties, such as network bandwidth and latency measurements. Several different resource discovery protocols and infrastructure exist that can discover new services and resources for a system. Likewise, effector technologies are emerging to support dynamic changes to running system components (e.g., Workflakes [16]). Also, for many legacy systems, it is conceivable to use wrappers to add hooks for making system changes.

Another assumption is that there are probes to provide information about any system properties and gauges to aggregate the information and update model properties. This assumption depends on the APIs of the gauges and probes being well-defined. We anticipate that there would be external developers who specialize in developing gauges and probes for various purposes. Furthermore, any of the measurement tools mentioned above may also implement the Probe API (or be wrapped) to serve as probes. Then developers of self-adapting systems can plug in any gauge and probe to suit their needs [13].

Finally, the work presented in this paper is inherently centralized, where monitoring and adaptation are performed in a centralized fashion within a single Rainbow instance. Making this assumption has allowed us to focus on core issues of self-adaptation, namely monitoring, detection, resolution, and adaptation. At the same time, there may be concerns of scalability and single point of failure. However, the Rainbow framework may be applied in a distributed setting. For example, one might apply instances of Rainbow to adapt multiple subsystems of a distributed system, then coordinate those instances toward an overall adaptation goal. The coordination and other distributed computing issues are future research problems.

5 Related Work

IBM's Autonomic Computing initiative categorizes self-managing systems as *self-configuring* (to adapt automatically to dynamically changing environments), *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 themselves from any attacks) [10]. Rainbow uses a software architecture approach to support *self-configuring* and *self-healing*.

The notion of architectural style has helped to capture units of reuse in software engineering at system design time. A notable example is an attribute-based architectural style (ABAS), which forms a reusable "pre-analyzed" software architecture part [19]. Our research extends this support of reuse to the context of run-time system self-adaptation.

Most closely related to this research is a collection of recent work that uses an externalized approach to dynamically monitor and adapt a running system. They focus on the use of specific styles (together with their associated ADLs and toolsets) to support architecture-based self-adaptation. For example, Taylor and colleagues use hierarchical publish-subscribe via C2 [8, 24]; Gorlick and colleagues use data-flow style via Weaves [15]; and Magee and colleagues use bi-directional communication links via Darwin and have even proposed a distributed self-organizing system where components coordinate toward a common architectural structure [20, 14]. In contrast, our framework is not bound to a particular architectural style and can potentially be applied to different styles of systems.

6 Conclusions and Future Work

In this paper we have demonstrated the generality of the Rainbow framework for architecture-based self-adaptation based on two case study systems. We showed that the adaptation infrastructure of the framework—including the measurement and resource discovery mechanisms; the model manager, constraint evaluator, and adaptation engine and executor; and the translation infrastructure—can be reused across different systems. We analyzed the case study systems to derive their styles and concerns, which are used to tailor the framework. We further derived guidelines to reason about when to reuse the system-specific adaptation knowledge. Since the concepts of styles and concerns are

fairly general, we believe that most systems can be analyzed for styles and concerns, and our framework can provide a low-cost approach to add self-adaptation to these systems.

Acknowledgments

The research described in this paper was supported by DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA.

References

[1] *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, Dec. 12–14 2001.

[2] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[3] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for self-repair. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development, and Maintenance*, pages 45–59, Montral, Qubec, Canada, Aug. 25–30 2002. Kluwer Academic Publishers.

[4] S.-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu. Software architecture-based adaptation for grid computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 389–398, Edinburgh, Scotland, 23–26 2002.

[5] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, editors. *Documenting Software Architecture: Views and Beyond*. The SEI Series in Software Engineering. Pearson Education, Inc., 2003.

[7] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A highly-extensible, XML-based architecture description language. In R. Kazman, editor, *Proceedings of WICSA2*, Amsterdam, The Netherlands, Aug. 28–31 2001. Kluwer Academic Publishers.

[8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. [11], pages 21–26.

[9] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the remos system. 2001.

[10] A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[11] D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, Charleston, SC, USA, Nov. 18–19 2002. ACM Press.

[12] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural descriptions of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge Univ Press, 2000.

[13] D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture* [1].

[14] I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architectures for distributed systems. In Garlan et al. [11], pages 33–38.

[15] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *13th International Conference of Software Engineering*, pages 23–34. IEEE, IEEE Computer Society, May 19981.

[16] P. N. Gross, S. Gupta, G. E. Kaiser, G. S. Kc, and J. J. Parekh. An active events model for systems monitoring. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture* [1].

[17] J.-C. Hu and J.-M. Ho. A Conference Gateway Supporting Interoperability Between SIP and H.323 Clients. Master's thesis, Carnegie Mellon University, Mar. 2000.

[18] A.-C. Huang and P. Steenkiste. Network-Sensitive Service Discovery. In *Proc. USITS '03 (to appear)*, Mar. 2003.

[19] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. In P. Donohoe, editor, *Proceedings of WICSA1*, pages 225–243, San Antonio, TX, USA, Feb. 22–24 1999. Kluwer Academic Publishers.

[20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proceedings of 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[21] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, Jan–Feb 1997.

[22] M. Moriconi and R. A. Reimenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, SRI International, 1997.

[23] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. *INFOCOM '02*, June 2002.

[24] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *20th International Conference of Software Engineering*, pages 177–186. IEEE, IEEE Computer Society, Apr. 19–25 1998.

[25] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248. ACM Press, 2002.

[26] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[27] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In Y. Deng and M. Gerken, editors, *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, pages 146–151. Knowledge Systems Institute, 1998.