In Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 15-19 2002.

Exploiting Architectural Design Knowledge to Support Self-repairing Systems

Bradley Schmerl and David Garlan
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213 USA
+1 412 268 5057
{schmerl,garlan}@cs.cmu.edu

ABSTRACT

In an increasing number of domains software is now required to be self-adapting and self-healing. While in the past such abilities were incorporated into software on a per system basis, proliferation of such systems calls for more generalized mechanisms to manage dynamic adaptation. General mechanisms have the advantage that they can be reused in numerous systems, analyzed separately from the system being adapted, and easily changed to incorporate new adaptations. Moreover, they provide a natural home for encoding the expertise of system designers and implementers about adaptation strategies and policies. In this paper, we show how current software architecture tools can be extended to provide such generalized dynamic adaptation mechanisms.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer Aided Software Engineering (CASE)*. D.2.11 [**Software Engineering**]: Software Architectures – *Languages*

General Terms

Design, Measurement.

Keywords

Software architecture, software architecture tools, dynamic adaptation, reflective systems.

1. INTRODUCTION

The increasing complexity of software means that there is more reliance on tools to capture knowledge and expertise about various aspects of the system, without requiring the tool user to understand the specific algorithms or formal methods underpinning this knowledge. For example, code-related tools may detect memory leaks or concurrency problems, requirements-tracing tools relate code to the requirements they fulfill, and software architecture tools capture design expertise about systems.

One of the key aspects of such tools is separating concerns in software to make change more manageable. For example, software architecture tools separate out the concern of the runtime topology and properties of the system. The expertise captured in the software architecture may specify topological or performance constraints on the system.

An increasingly important aspect of software is providing mechanisms for runtime adaptation. In the past, the requirement for dynamically changing the system was restricted to certain categories of applications, where human oversight was impossible or difficult (such as satellite software), or where the cost of stopping the system to perform maintenance was prohibitive (such as in telecommunication systems). However, this requirement is becoming increasingly important in general distributed systems, defense software, and pervasive computing. Even desktop applications and operating systems are beginning to allow upgrades without shutting systems down.

Even though the requirement for self-adaptation is becoming more widespread, the mechanisms to implement this requirement are still mostly provided in a per-system, ad hoc manner, with the adaptation typically embedded in the application code. Consequently, knowledge about self-adaptation is difficult to separate from application code, making adaptation knowledge difficult to (a) transfer or reuse in other applications, (b) reason about or analyze to ascertain whether the adaptations will result in proper-running systems, and (c) change. Furthermore, because adaptation is distributed among the various components or modules in the code, the adaptation performed tends to be localized to those components. It does not work well for global adaptation (e.g., requiring changes in the structure of the application) or in cases where multiple adaptation operations have to be coordinated. As such, tools to support and implement reusable, analyzable, and global self-adaptation mechanisms are scarce and are limited to specific domains.

Recently a number of researchers have proposed an alternative approach in which system models – and in particular, architectural models – are maintained at run time and used as a basis for system reconfiguration and repair [15]. Architecture-based adaptation has a number of nice properties: (1) as an abstract model, an architectural model can provide a global perspective on

the system; (2) architectural models can make "integrity" constraints explicit, helping to ensure the validity of any change; (3) suitably-designed architectures permit flexible evolution of systems by providing loose coupling between components; and (4) architectures encapsulate the design knowledge, and thereby provide a basis for principled adaptation.

Although attractive in principle, there are a number of roadblocks to making software architecture useful as the basis for runtime adaptation. Most importantly, today's tools and representations are intended as development-time aids. Consequently, the expertise captured in the architecture cannot be easily exploited to help manage run-time change.

In other work [4,5] we describe how the Acme architecture description language (ADL) can be extended to support runtime adaptation. In this paper, we consider how one can reuse existing design-time tools to help with to support runtime architectural adaptation, and what additional infrastructure and components are needed to fill out the picture. We then detail our experience in adapting the Acme toolsuite in this way.

2. RELATED WORK

Research in software architecture has matured in the last decade to the point that there is a plethora of tool support for various aspects of architectural design. This research has mainly focused on design-time support for constructing and analyzing architectures, with formal knowledge embodied in concomitant tool support. There are almost as many architecture definition tools as there are ADLs, and a wide variety of analyses can be performed on architectures. A summary of some of this research is presented in [13]. In this section we focus on the different tools for relating architectures to code.

Within particular domains, there are a number of tools that provide some support for generating code, or code stubs, from architectures. For example, C2 [18], Unicon [16], and Weaves [10] provide implementations and libraries for specific architectural domains.

The Rapide toolset [12] allows the creation of an executable simulation of an architecture, and provides various analysis tools to replay the events according to the architecture and check the conformance of the simulation to formal constraints in the architecture. However, this analysis and tools are based on simulations and are not run concurrently with the executing program to either monitor or change the running system.

The crucial missing piece in the above work is providing generalized architectural adaptation that does not assume a particular style and runs concurrently with the executing system. In other work, we have shown how to generalize architecture-based adaptation by making the choice of architectural style an explicit design parameter in the adaptation framework [4], and have reported results of some experiments that we have conducted within this framework for a particular kind of architecture [5]. In this paper, we focus on the use of software architectural tools to aid in constructing run-time adaptation mechanisms. In particular, we detail our experience with tools that support the Acme architectural description language.

3. SOFTWARE ARCHITECTURE

The centerpiece of our approach is the use of architectural models [1,7]. We use a simple scheme in which an architectural model is represented as a graph of interacting components. This is the core architectural representation scheme adopted by a number of architecture description languages, including Acme [8], xADL [6], and SADL [14]. 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.

To account for various behavioral properties of a system, elements in the graph can be annotated with property lists. For example, properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). The software architecture can also specify a set of constraints that must be maintained. Constraints can, for example, specify that some property value must always be within a certain range. One of the advantages of architectural descriptions is that they provide opportunities for automatic verification of such constraints.

Representing an architecture as an arbitrary graph of generic components and connectors has the advantage of being extremely general and open ended. However, in practice there are a number of benefits to constraining the design space for architectures by associating a *style* with the architecture. An architectural style typically defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed. Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [7,18,19].

4. OVERVIEW OF OUR APPROACH

In order to exploit architectural design knowledge for runtime adaptation, we must support three activities that occur in a control-loop fashion. The three activities are:

Monitoring: The first element of determining whether the system needs to be adapted is to observe its runtime behavior. These observations can be made by various technologies that can determine information about the system. Examples of such technologies are code instrumentation to report the occurrence of specific method calls [2], and operating system and network monitoring systems to measure performance [11]. The key idea here is that at this level, monitoring of the application is in terms that are relevant to the implementation – they are likely to be localized, system-specific observations and mechanisms.

Interpretation: The information observed about the running system in its raw form is not likely to be suitable for analysis at the architectural level – there needs to be some interpretation of the data that reflects these observations in the context of a higher-level architectural model. For example, if connector protocols are specified at the architectural level, then method calls in the implementation will need to be interpreted and reported as events in the protocol. Similarly, low-level system performance observations may need to be interpreted as aggregate throughputs or latencies at the architectural level. This interpretation is crucial so that the observed properties can be used in architectural analysis

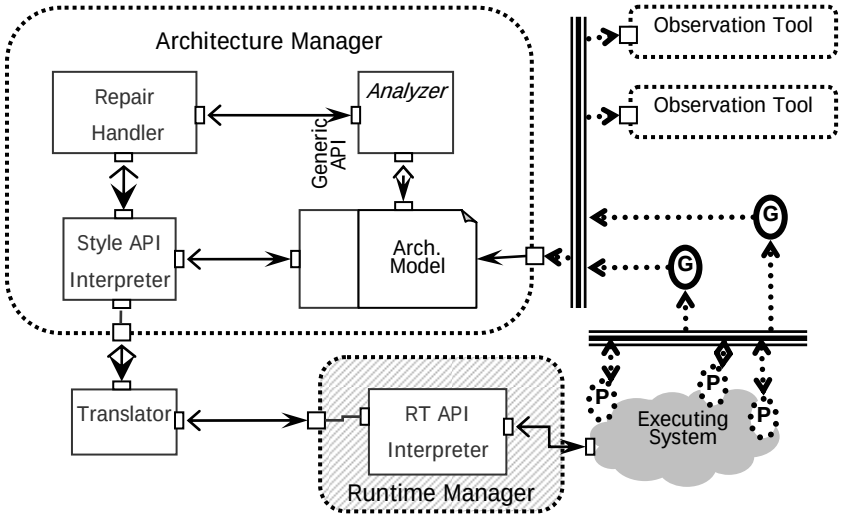

Figure 1. Adaptation Framework.

to ascertain whether there is something wrong with the system. Our contention is that a high-level architectural model can be more easily reasoned about than the runtime-state of the system and its implementation code. The methods used for this interpretation are discussed in more detail in [9].

Reconfiguration: Once the runtime analysis of the architectural model has been conducted, using information reported by the interpretation activity, it is possible to determine whether the architectural model is correct according to its style, and also with respect to analyses facilitated by that style. If the architectural model is not consistent with its design assumptions, then there is a need to reconfigure the system. Such reconfigurations are intended to adapt the architecture so that it returns to a consistent state, and also to make the concomitant changes in the implementation. For our system, reconfiguration at the architectural level is done via repair strategies, as discussed in [4]. Reflecting those changes to the implementation in a general way is the subject of ongoing research.

4.1 Adaptation Framework

To determine the role of tool support for these activities, we have developed a framework that is similar to [15] (as illustrated in Figure 4): An executing system is monitored to observe its run time behavior via a set of probes (indicated by **P**'s in the figure). Monitored values are abstracted and related to architectural properties of an architectural model (**G**'s in the figure). When a property in the architectural model changes, constraints are reevaluated to determine whether the system is operating within an envelope of acceptable ranges (*Analyzer*). Violations of constraints are handled by a repair mechanism (*Repair Handler*), which adapts the architecture. Architectural changes are propagated to the running system via the *Translator* and the *RTI API Interpreter*.

As we detail later, the key new feature in this framework is the use of style as a first class entity that determines the actual behavior of each of the parts. Specifically, style is used to determine (a) what properties of the executing system can monitored, (b) what analysis needs to be conducted, (c) what to do when an error is detected, and (d) how to carry out repair in terms of style-based architectural operators. In addition we need to introduce a style-specific translation component to manage the transactional nature of repair and map high-level architecture operations into lower-level system operations.

5. ACME ARCHITECTURE DESIGN TOOLS

The framework described above makes the use of architectures and architectural styles explicit, and is suitable for use with an ADL and its associated tools. The Acme ADL was originally conceived as an interchange language between numerous, style-specific ADLs. Extended with types, constraints, and families, it has evolved into a powerful ADL in its own right. Several tools that use Acme as their native ADL have been developed, with our research focusing on the support for style-related architectural analysis.

Tool support for Acme has coalesced on two fronts: the definition of an architectural model, and the analysis of the model once it has been defined. AcmeStudio, developed at Carnegie Mellon University, and the Acme PowerPoint Editor, developed at ISI, are two such tools that provide a graphical design environment based on Acme for defining architectures. Armani, performance analysis, and various style-specific analysis tools have been developed that calculate properties of an architectural model and evaluate satisfaction of constraints.

5.1 The AcmeStudio Design Environment

AcmeStudio is a graphical design tool written in Visual C++™ for Windows™ platforms. It uses Acme as its core language and takes advantage of Acme styles (called *families*) to allow the definition of custom visualizations, rudimentary property-based analysis for visualization variants, and the types of analysis that can be performed. Figure 2 illustrates a session with AcmeStudio that contains two architectural models (window panes for editing

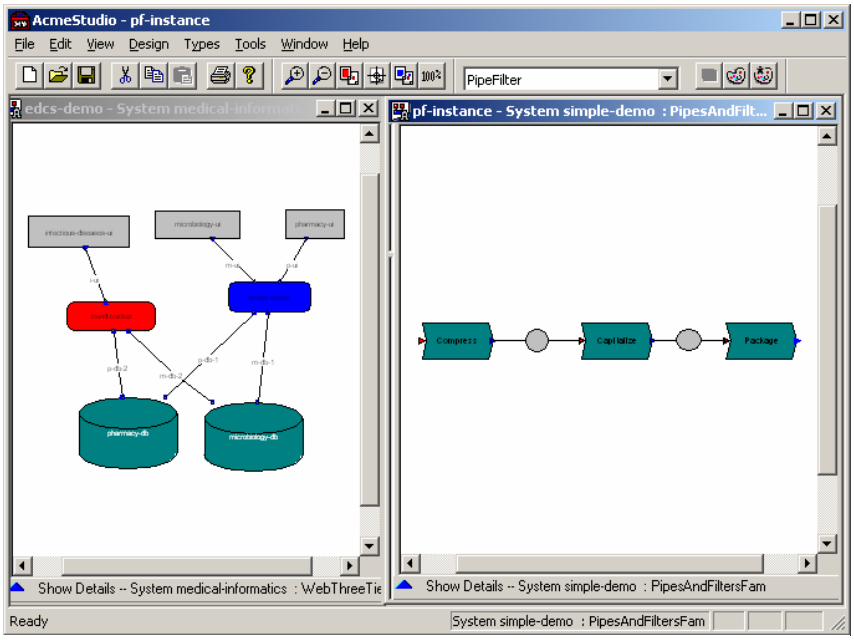

Figure 2. AcmeStudio.

and additional information have been hidden for space reasons). On the left of the figure is an architectural model of a three-tiered web client-server system, and on the right is a model of a pipe and filter architecture.

AcmeStudio uses Acme families to provide both the types that can be used in the architectural model and the way those types should be depicted. For example, in the figure, components of type FilterT are depicted as arrowed boxes and connectors of type PipeT as gray circles. The ports and roles in the architecture are depicted as small triangles. In the left hand architecture, components of type RepositoryT are shown as repository icons. The key idea here is to use Acme families not only to define the vocabulary of the architecture, but also to use it as the basis for visualization. In this way, architects can use the style of diagram they desire or are accustomed to.

In addition to depictions based on element type, AcmeStudio incorporates the concept of visual variants. A visual variant is a variation on the depiction of a type based on values of an instance's properties. Architects can use this feature to, for example, flag errors in the architecture. The middle level components in the architecture on the left side of Figure 2 illustrate this concept. A variant specifies that if a server is overloaded, it should be depicted with a different fill color (light gray in the figure[1]). In Figure 2, we have conducted a performance analysis based on queuing theory from AcmeStudio [17]. This has set various properties of the components and connectors in the architecture. One of these properties is the Boolean property sOverloaded. The left-most server in the figure is overloaded, and so the visualization has changed.

In addition to the core functionality of graphical design, AcmeStudio can be extended with additional functionality via COM based tools. Such tools can be used to provide domain-specific analyses of the architecture, and report the results of the analysis back to AcmeStudio.

5.2 Armani Constraint Analysis

One such COM tool is the Armani constraint analysis tool. Armani is a first-order predicate language extension to Acme that can be used to analyze structural and other properties of the architecture. For example, one of the constraints on a filter in the Pipe and Filter family is:

```
invariant forall p : port in self.ports |
    declaresType (p, WritePortT) or declaresType (p, ReadPortT)
```

This constraint states that each port in a filter must either be a write port or a read port. In such a fashion, architects can prevent designers from adding incorrect ports to components. Other constraints might specify that the pipe and filter system cannot have any cycles. The WebThreeTier family might specify that each client must be connected to a server.

When this tool is invoked from AcmeStudio, the constraints in the architectural model are evaluated by the Armani tool. Errors are reported back to AcmeStudio. Error reports include the constraint that failed and the architectural element(s) over which it failed. This information is used by AcmeStudio to allow easy identification and navigation to erroneous parts of the architectural model.

5.3 Architectural Tools for Dynamic Change

So far, we have discussed the tools that have been used for architectural design, and given details of some design-time Acme tools that are used to construct and analyze architectures. If we are using software architectural models and analyses to guide dynamic adaptation, then it is useful to use these tools at runtime.

[1] In actual fact, the color is changed to red.

This approach preserves continuity between design time and runtime views of the system, and maintains uniformity of the types of analyses that are performed at runtime and their meaning with respect to the design-time architectural artifacts.

Given that we want to use existing architectural tools at runtime, the question arises as to what role they should play in runtime adaptation, how they should be adapted to be used in the dynamic context, and what additional tools are required. The guiding principle in providing tool support under this framework is that the separation of concerns that exist in the framework outlined earlier should be maintained. This separation of concerns is:

- the use of different architectural views and analyses, that may reside in several design-time tools;
- the ability to monitor different attributes of the architecture at different times in execution;
- the desire to employ different types of repair strategies in the framework; and
- the fact that there may be many mappings from a particular architectural model, expressed in a particular architectural style, to an implementation of that system.

Separating these concerns allows the different kinds of expertise required for dynamic adaptation into different appropriate tools, rather than attempting to develop a monolithic tool to perform all aspects of adaptation. Thus, in the design of our toolset we have modified our existing design-time tools to observe and analyze the architecture, and developed new tools to capture the knowledge particular to each concern. While we discuss this with respect to some Acme-based architecture tools, we believe that analogous modifications will need to be made to any architecture tool to fit into the general adaptation framework of Figure 1.

5.3.1 Changes to Existing Tools

The changes to the existing toolset fall broadly into the following categories:

- Interfaces that allow the architectural model to be changed dynamically.
- Integration points between architectural analysis tools and facilities to effect a repair, should analysis determine something is wrong.
- Facilities to allow a designer to indicate points in the architecture that should be monitored, and the types of monitoring that should be conducted.
- Handling associated scalability issues of runtime analysis, in reaction to observations of the executing system.

We show how we addressed these categories in the case of AcmeStudio and Armani.

AcmeStudio: The role of AcmeStudio in the dynamic adaptation framework is twofold. First, it is still used at design time to define the architecture. For this stage, AcmeStudio has been extended to allow gauges to be attached to points in the architecture. Once again, this is based on families – families define which gauge types are available to a system. If a family defines such gauge types, instances can be dropped onto the design and attached to properties in the architecture. For example, to request monitoring of the latency of a particular connector in the architectural model, a designer would (1) ensure there is an appropriate property of the connector (for example, latency); (2) add a gauge to monitor connector latency to the architectural model; and (3) bind the value reported by that gauge to the property in the connector.

The second role of AcmeStudio is as an observation tool in the adaptation framework. Once the system is started, AcmeStudio is no longer used to edit the architecture – it merely observes the changes made to the architecture by other tools. A new tool extracts gauge information from the architectural model and creates the requested gauges. AcmeStudio has been extended with a COM interface through which gauges can report changing property values.

The AcmeStudio COM interface also contains routines to change the architectural model – create, delete, or modify components, connectors, etc. In this way, tools that do the actual analysis and modification can inform AcmeStudio, so that the changed architecture can be viewed. For example, if a gauge detects an overloaded server, it can report this fact as the sOverloaded property of the corresponding architectural component. AcmeStudio, using existing visual variants, can change the component color to light gray.

Armani Constraint Analysis: Armani has been extended with an imperative language that can be used to define repair strategies to programmatically change the architecture. A *repair strategy* can optionally be associated with an Armani constraint. If the constraint fails during runtime analysis, then the repair strategy is invoke. A repair strategy is composed of a number of subsidiary constraints and repair tactics. This allows a repair strategy to conduct more than one change, based on further investigation of the problem. For example, if an Armani constraint specifying that latency must be below a certain threshold is violated, the repair strategy will likely contain tactics to address the case if the bandwidth has fallen or the load on servers has risen. Furthermore, repair strategies contain decision logic for choosing which of the tactics to apply.

An example repair strategy is presented in Figure 3. The Armani constraint and repair strategy to invoke are shown in lines 1-3 of the figure. In line 2, "!→" is a new operator that specifies that the repair strategy following is to be executed only if the constraint is violated. The top-level repair strategy in lines 5-17, fixLatency, consists of two tactics, only one of which is chosen to be executed by this repair strategy. The first tactic in lines 19-31 handles the situation in which a server group is overloaded, identified by the precondition in lines 24-26. Its main action in lines 27-29 is to create a new server in any of the overloaded server groups. The second tactic in lines 33-48 handles the situation in which high latency is due to communication delay, identified by the precondition in lines 34-36. It queries the architecture to find a server group that will yield a higher bandwidth connection in lines 40-41. In lines 42-44, if such a group exists it moves the client-server connector to use the new group.

```
01 invariant r.Avg_Latency <= maxLatency
02 !→
03     fixLatency(r);
04
05 strategy fixLatency (badRole: ClientRoleT) = {
06     begin repair-transaction;
07     let badClient: ClienT =
08         select one cli: ClientT in self.Components |
09             exists p: RequestT in cli.Ports | attached(badRole, p);
10     if (fixServerLoad(badClient)) {
11         commit repair-transaction;
12     else if (fixBandwidth(badClient, badRole) {
13         commit repair-transaction;
14     } else {
15         abort(ModelError);
16     }
17 }
18
19 tactic fixServerLoad (client: ClientT) : boolean = {
20     let overloadedServerGroups: Set{ServerGroupT} =
21         { select sgrp: ServerGroupT in self.Components |
22             connected(sgrp, client) and
23             sgrp.Server_Load > maxServerLoad };
24     if (size(overloadedServerGroups) == 0) {
25         return false;
26     }
27     foreach sGrp in overloadedServerGroups {
28         sGrp.addServer();
29     }
30     return (size(overloadedServerGroups) > 0);
31 }
32
33 tactic fixBandwidth (client: ClientT, role: ClientRoleT) : boolean = {
34     if (role.Bandwidth >= minBandwidth) {
35         return false;
36     }
37     let oldSGrp: ServerGroupT =
38         select one sGrp: ServerGroupT in self.Components |
39             connected(client, sGrp);
40     let goodSGrp: ServerGroupT =
41         findGoodSGrp(client, minBandwidth);
42     if (goodSGrp != nil) {
43         client.moveClient(oldSGrp, goodSGrp);
44         return true;
45     } else {
46         abort(NoServerGroupFound);
47     }
48 }
```

Figure 3. An Example Repair Strategy.

In addition to extending the Armani language, we are investigating ways to optimize the performance of the constraint analysis at runtime with incremental approaches.

5.3.2 New Architecture Tools

The existing tools address the concerns of observation and analysis in our framework. However, they do not address how to implement monitoring, how to execute the repair, or how to map between an architectural model and its implementation. These new tools are now discussed.

Gauge Infrastructure: As illustrated in Figure 1, gauges are used to do abstraction and propagate information about the runtime system to the architectural model. We have developed a gauge infrastructure, implemented as a Java class library that provides implementation stubs for gauges, and routines to communicate between gauges and tools that consume gauge outputs [9]. Because of the requirement for working in distributed systems, we have implemented the transport layer of the gauge infrastructure using the Siena wide area event notification system [3].

Tailor Repair: In concert with the repair extension to Armani, we are developing tools that provide runtime execution of these repairs. The goal of Tailor is to execute repairs that return an erroneous architecture to one that conforms to its style and constraints. Tailor listens to gauges for values associated with the model it is trying to maintain. It then invokes Armani to check if any constraints are violated. If they are, it executes the appropriate repair tactics. Tailor is decoupled from the executing system, and can run on a machine independent of the running system. In this way, we anticipate that monitoring and repair at the architectural level will not unduly impede the running system.

Mapping Between Architecture and Implementation: Currently in our toolset we assume that gauges provide a mapping between runtime observations and architectural observations. In fact, this is just one example of mapping that is required throughout the framework. For our approach to be effective, we require a two-way mapping between information in the runtime system and information in the architecture. Both directions are required by Tailor. A mapping from the implementation to the architecture is required when Tailor investigates the state of the running system to determine the best tactic. (For example, in Figure 3 Tailor needs to determine to which server group to move a client.) The mapping from the architecture to the runtime system is required when Tailor issues architectural changes that need to be reflected in the implementation. For example, in Figure 3, Tailor issues the architectural repair addServer (line 28), which needs to be translated to starting a server process on a particular host and joining a particular server group.

We do not assume that the mapping between architecture and implementation is one-to-one. Indeed, a particular architectural style, for example a client server architecture, could be associated with many "implementation styles." Currently, this information is captured in the Translator component of our framework and we are investigating methods of generalizing this component so that we can specify the transformations for multiple styles. Once this component is in place, it could also be used by gauges to associate runtime observations with architectural properties, in contrast to our current implementation, which embeds this information in the gauges themselves.

6. Integrating the Architectural Tools

The development of different tools to capture specific knowledge about different aspects of dynamic adaptation means that these tools need to be integrated in some fashion. The framework in Figure 1 gives a broad outline of how to do this.

Figure 4 provides an illustrative example of how we have integrated our tools, in the particular case of adapting a client-server system. The running distributed client-server system is on the left of the figure, and consists of three clients, three servers, and a request queue component. Clients make requests to the request queue and servers serve requests that they pull from the request queue. To instrument this system, each component is run inside an AIDE shell [2], which allows us to probe the method calls inside the component. This implementation corresponds to the

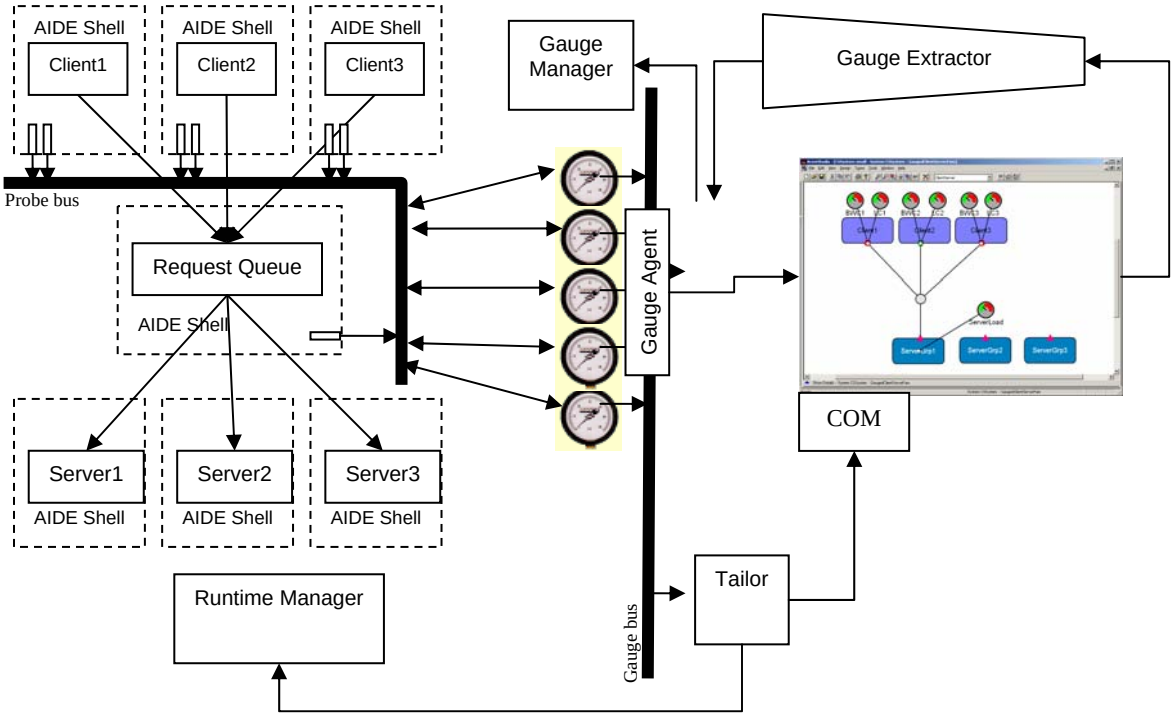

Figure 4. Runtime Adaptation Example

example reported in [4], which calls for adaptation if the latency rises above 2 seconds. Using our tools to effect the adaptation requires several steps. (We assume that the system is running, and that the architecture for this system is already defined.)

The first step is to use AcmeStudio to attach gauges to various properties in the architecture. In the client-server example, we attach gauges to the server load property of the request queue component of the architecture, and two gauges to each of the client roles in the architecture – one to report the bandwidth and one to report the average latency experienced by clients attached to the role.

The next step involves starting system monitoring by starting gauges. AcmeStudio invokes the *Gauge Extractor* tool, which communicates via RMI with a *Gauge Agent*. The Gauge Agent is the mediator between gauges and AcmeStudio.

1. The Gauge Agent locates *Gauge Managers* to start particular gauges and then creates the required gauges (in the middle of the figure).
2. These gauges create the necessary implementation probes. The probes in this example report whenever the newRequest method is called in a client. Probes report the size of the data contained in response corresponding to the request. A probe in the Request Queue reports the size of the queue.
3. The gauges interpret this low-level, method-call information into high level latency and bandwidth values and report these values to the gauge bus.
4. The Gauge Agent reports gauge values to AcmeStudio, which can display the results.
5. Concurrently, Tailor listens to the gauge bus and evaluates Armani constraints to determine if the system is still performing acceptably. If not, it makes changes to its internal model of the architecture and reports these changes to AcmeStudio, via the COM interface, and the *Runtime Manager*, via RMI.
6. The Runtime Manager in this example contains a simple table-based mapping between architectural changes and runtime changes, and performs the necessary changes to the running system based on the repair tactic chosen by Tailor.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how to adapt existing design-time architecture tools so that the knowledge encapsulated within them is made available to adapt executing systems dynamically, and have shown how we have adapted our Acme tool suite for this task. Furthermore, we identified and outlined the implementation and integration of additional tools for architecture-based monitoring and repair of an executing system. The use of architectural tools for runtime adaptation is desirable because it (a) allows the use of design time architectural expertise to guide adaptation, (b) provides a continuity of viewpoint from the design of the system to its dynamic behavior, and (c) allows us to use analysis, previously available only at design time, to effect changes based on the dynamic behavior of the system.

The framework and tools described in this paper provide a clean separation of concerns into distinct aspects of monitoring, analysis, repair, and translation between architecture and implementation. Thus, it provides a platform for experimenting with various approaches to each of these aspects.

While we have focused on how we adapted Acme tools to fit into this framework, there are some general lessons about how to integrate design-time architectural tools into a dynamic adaptation framework:

1. The tool needs to have a runtime interface that allows models to be changed on the fly.
2. Monitors need to be written to observe the running system and relate these observations to the architectural model described in the tool.
3. The architectural model needs to be dynamically re-evaluated, taking into consideration these observations.
4. The results of reevaluation need to be related to repairs that can be made to the architectural model and the running system, and these repairs need to be executed dynamically.

Future work involves further elaboration and development of our Acme toolsuite to this framework, and the development of tools for each of the concerns outlined above. In particular, we are investigating tools for easing the development of gauges, and tools for facilitating the capture of knowledge about the association between an architectural model and an implementation.

ACKNOWLEDGEMENTS

This work is supported in part 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] Allen, R.J. A Formal Approach to Software Architecture. PhD Thesis, published as Carnegie Mellon University School of Computer Science Technical Report CMU-CS-97-144, May 1997.

[2] Calnan, P. Semantic-based Code Transformation. MS Thesis Proposal, Department of Computer Science, Worchester Polytechnic Institute, Massachusetts, March 2002.

[3] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, July, 2000.

[4] Cheng, S.-W., Garlan, D., Schmerl., B., Sousa, J., Spitznagel, B., Steenkiste, P. Using Architectural Style as the Basis for Self-repair. The Working IEEE/IFIP Conference on Software Architecture 2002, Montreal, August 25-31, 2002.

[5] Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J., Spitznagel, B., Steenkiste, P. Software Architecture-based Adaptation for Grid Computing. Proc. the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, July 2002.

[6] Dashofy, E., van der Hoek, A., Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. Proc. 2nd Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 2001.

[7] Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. Proc. SIGSOFT '94 Symposium on the Foundations of Software Engineerng, , New Orleans, LA, December 1994.

[8] Garlan, D., Monroe, R.T., and Wile, D. Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems. Leavens, G.T., and Sitaraman, M. (eds). Cambridge University Press, 2000 pp. 47-68.

[9] Garlan, D., Schmerl, B., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. Proc. Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 12-14 December, 2001.

[10] Gorlick, M.M., and Razouk, R.R. Using Weaves for Software Construction and Analysis. Proc. 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991.

[11] Lowekamp, B., Miller, N., Sutherland, D., Gross, T., Steenkiste, P., and Subhlok, J. A Resource Query Interface for Networ-aware Applications. Cluster Computing, 2:139-151, Baltzer, 1999.

[12] Luckham, D.C. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.

[13] Medvidovic, N., and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, January 2000.

[14] Moriconi, M. and Reimenschneider, R.A. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI International, March 1997.

[15] Oriezy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3):54-62, May/June 1999.

[16] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., and Zelesnik, G. Abstractions for Software Architecture and Tools to Support them. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4):314-335, April, 1995.

[17] Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proc. the 1998 Conference on Software Engineering and Knowledge Engineering, June, 1998.

[18] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6):390-406, 1996.

[19] Vestel, S. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.