Submitted for publication.

Exploiting Architectural Style for Self-repairing Systems

Shang-Wen Cheng, David Garlan, Bradley Schmerl
João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213 USA
+1 412 268 5056
{zensoul,garlan,schmerl,jpsousa,sprite,prs}@cs.cmu.edu

ABSTRACT
An increasingly important requirement for software systems is the capability to adapt at run time in order to accommodate varying resources, system errors, and changing requirements. For such self-repairing systems, one of the hard problems is determining when a change is needed, and knowing what kind of adaptation is required. In this paper we describe a partial solution in which stylized architectural design models are maintained at run time as a vehicle for automatically monitoring system behavior, for detecting when that behavior falls outside of acceptable ranges, and for deciding on a high-level repair strategy. The main innovative feature of the approach is the coupling of architectural models with architectural styles that both exposes specific properties of interest and provides an analytic basis for detecting anomalies and suggesting verifiably sound repair strategies. We illustrate the approach for performance-oriented adaptation of web-based client-server applications, whose soundness can be determined through queuing theoretic analysis of the corresponding architectures.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Domain-specific architectures, Languages.

General Terms
Measurement, Performance, Design, Theory.

Keywords
Dynamic adaptation, software architectures, performance analysis.

1. INTRODUCTION
An increasingly important requirement for software-based systems is the ability to adapt themselves at run time to handle such things as resource variability, changing user needs, and system faults. In the past, systems that supported self-adaptation were rare, confined mostly to domains like telecommunications switches or deep space control software, where taking a system down for upgrades was not an option, and where human intervention was not always feasible. However, today more and more systems have this requirement, including e-commerce systems and mobile embedded systems. Such systems must continue to run with only minimal human oversight, and cope with variable resources (bandwidth, server availability, etc.), system faults (servers and networks going down, failure of external components, etc.), and changing user priorities (high-fidelity video streams at one moment, low fidelity at another, etc.).

Traditionally system self-repair has been handled within the application, and at the code level. For example, applications typically use generic mechanisms such as exception handling or timeouts to trigger application-specific responses to an observed fault or system anomaly. Such mechanisms have the attraction that they can trap an error at the moment of detection, and are well-supported by modern programming languages (e.g., Java exceptions) and run time libraries (e.g., timeouts for RPC). However, they suffer from the problem that it can be difficult to determine what the true source of the problem is, and hence what kind of remedial action is required. Moreover, while they can trap errors, they are not well-suited to recognizing "softer" system anomalies, such as gradual degradation of performance over some communication path.

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 [27]. Architecture-based adaptation has a number of nice properties: As an abstract model, an architecture can provide a global perspective on the system. Architectural models can make "integrity" constraints explicit, helping to ensure the validity of any change. Suitably-designed architectures permit flexible evolution of systems by providing loose coupling between components.

A key issue in making this approach work is the choice of architectural style used to represent a system. Previous work in this area has focused on the use of specific styles (together with their associated ADLs and toolsets) to provide intrinsically modifiable architectures. Taylor and colleagues use hierarchical publish-subscribe via C2 [26, 29]; Gorlick and colleagues use dataflow style via Weaves [12]; and Magee and colleagues use bi-directional communication links via Darwin [18].

The specialization to particular styles has the benefit of providing strong support for adapting systems built in those styles. However, it has the disadvantage that a particular style may not be appropriate for an existing implementation base, or it may not expose the kinds of properties that are relevant to adaptation. For example, different styles may be appropriate depending on whether one is using existing client-server middleware, Enterprise JavaBeans (EJB), or some other implementation base. Moreover, different styles may be useful depending on whether adaptation should be based on issues of performance, reliability, or security.

In this paper we show how to generalize architecture-based adaptation by making the choice of architectural style an explicit design parameter in the framework. This added flexibility allows system designers to pick an appropriate architectural style in order to expose properties of interest, provide analytic leverage, and map cleanly to existing implementations and middleware.

The key technical idea is to make architectural style a first-class run time entity. As we will show, formalized architectural styles augmented with certain run time mechanisms provide a number of important capabilities for run time adaptation: (1) they define a set of formal constraints that allow one to detect system anomalies; (2) they are often associated with analytical methods that suggest appropriate repair strategies; (3) stylistic constraints can be linked with repair rules whose soundness is based on corresponding (style-specific) analytical methods; (4) they provide a set of operators for making high-level changes to the architecture; (5) they prescribe what aspects of a system need to be monitored.

In the remainder of this paper we detail the approach, focusing primarily on the role of architectural styles to interpret system behavior, identify problems, and suggest remediation. To illustrate the ideas we describe how the techniques have been applied to self-repair of an important class of web-based client-server systems, based on monitoring of performance-related behavior. As we will show, the selection of an appropriate architectural style for this domain permits the application of queuing-theoretic analysis to motivate and justify a set of repair strategies triggered by detection of architectural constraint violations.

2. RELATED WORK

Considerable research has been done in the area of dynamic adaptation at an implementation level. There are a multitude of programming languages and libraries that provide dynamic linking and binding mechanisms, as well as exception handling capabilities (e.g., [8, 14, 15, 23]) Systems of this kind allow system self-repair to be programmed on a per-system basis, but do not provide external, reusable mechanisms that can be added to systems in a disciplined manner *per se*, as with an architecture-driven approach.

Our work is also related to distributed debugging systems [13]. However, those systems have focused on user-mediated monitoring, whereas our research is primarily concerned with automated monitoring and reconfiguration.

Most closely related is the research on architecture-based adaptation, mentioned earlier. As we noted, the primary difference between our work and earlier research in this area is the decoupling of style from the system infrastructure so that developers have the flexibility to pair an appropriate style to a system based on its implementation and the system attributes that should drive adaptation. To accomplish this we have to introduce some new mechanisms to allow "run time" styles to be treated as a design parameter in the run time adaptation infrastructure. Specifically, we must show how styles can be used to detect problems and trigger repairs. We must also provide mechanisms that bridge the gap between an architectural model and an implementation – both for monitoring and for effecting system changes. In contrast, for systems in which specific styles are built-in (as with [12, 29]) this is less of an issue because architectures are closely coupled to their implementations *by construction*.

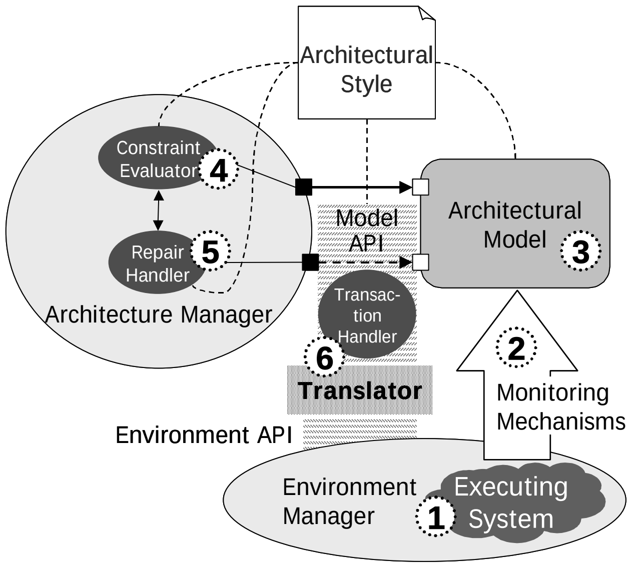

Figure 1. Adaptation Framework

Finally, there has been some work on formally characterizing architecture styles, and using them as a basis for system analysis [9, 28]. Our research extends this by showing how to turn "style as a design time artifact" into "style as a run time artifact". As we will see this requires two significant additions to the usual notion of style as a set of types and constraints: (1) style-specific repair rules, and (2) style-specific change operators. Some other efforts in this area have investigated formal foundations for this in terms of graph grammars and protocols, but have not carried the results through to implementation [2, 19, 31].

3. OVERVIEW OF APPROACH

Our starting point is an architecture-based approach to self-adaptation, similar to [27] (as illustrated in Figure 1): An executing system (1) is monitored to observe its run time behavior (2). Monitored values are abstracted and related to architectural properties of an architectural model (3). Changing properties of the architectural model trigger constraint evaluation (4) to determine whether the system is operating within an envelope of acceptable ranges. Violations of constraints are handled by a repair mechanism (5), which adapts the architecture. Architectural changes are propagated to the running system (6).

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 should be monitored, (b) what constraints need to be evaluated, (c) what to do when constraints are violated, and (d) how to carry out repair in terms of high-level 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.

To illustrate how the approach works, consider a common class of web-based client server applications that are based on an architecture in which web clients access web resources by making requests to one of several geographically distributed server groups (see Figure 2). Each server group consists of a set of replicated servers, and maintains a queue of requests, which are handled in FIFO order by the servers in the server group. Individual servers send their results back directly to the requesting client.

The organization that manages the overall web service infrastructure wants to make sure that two inter-related system qualities are maintained. First, to guarantee quality of service for the customer, the request-response latency for clients must be under a certain threshold (e.g., 2 seconds). Second, to keep costs down, the set of currently active servers should be kept as loaded as possible, subject to the first constraint.

Since access loads in such a system will naturally change over time, the system has two built-in low-level adaptation mechanisms. First, we can activate a new server in a server group or deactivate an existing server. Second, we can cause a client to shift its communication path from one server group to another.

The challenge is to engineer things so that the system adapts appropriately at run time. Using the framework described above, here is how we would accomplish this. First, given the nature of the implementation, we decide to choose an architectural style based on client-server in which we have clients, server groups, and individual servers, together with the appropriate client server connectors (see Figure 3). Next, because performance is the key quality attribute of concern, we adapt that style so that it exposes performance related properties and makes explicit constraints about performance (see Figure 4). Here, client-server latency and server load are the key properties, and the constraints are derived from the two desiderata listed above. Furthermore, because of the nature of communication we are able to pick a style for which formal performance analyses exist – in this case M/M/m-based queuing theory.

To make the style useful as a run time artifact we now augment the style with two specifications: (a) a set of style-specific architectural operators, and (b) a collection of repair strategies written in terms of these operators associated with the style's constraints. The operators and repair strategies are chosen based on an examination of the analytical equations, which formally identify how the architecture must change in order to affect certain parameters (like latency and load).

There are now only two remaining problems. First, we must get information out of the running system. To do this we employ low-level monitoring mechanisms that instrument various aspects of the executing system. We can use existing off-the-shelf performance-oriented "system probes," which we detail later. To

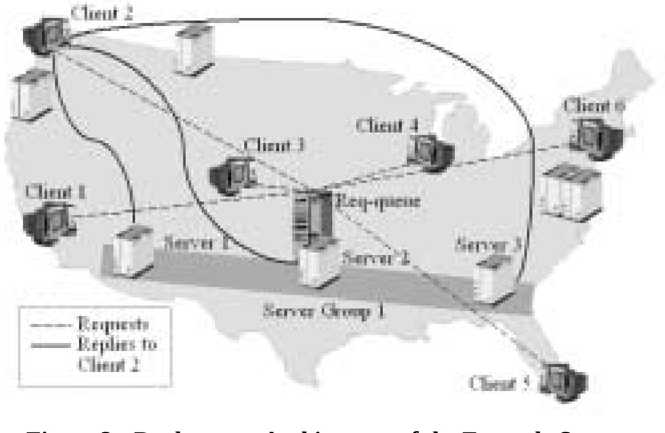

Figure 2. Deployment Architecture of the Example System.

bridge the gap between low-level monitored events and architectural properties we use a system of adapters, called "gauges," which aggregate low-level monitored information and relate it to the architectural model. For example, we have to aggregate various measurements of the round-trip time for a request and the amount of information transferred to produce bandwidth measurements at the architectural level.

The second problem is to translate architectural repairs into actual system changes. To do this we write a simple table-driven translator that can interpret architectural repair operators in terms of the lower level system modifications that we listed earlier.

In the running system the monitoring mechanisms update architectural properties, causing reevaluation of constraints. Violated constraints (high client-server latencies, or low server loads) trigger repairs, which are carried out on the architectural representation, and translated into corresponding actions on the system itself (adding or removing servers, and changing communication channels). The existence of an analytic model for performance (M/M/m queuing theory) helps guarantee that the specific modification operators for this style are sound. Moreover, the matching of the style to the existing system infrastructure helps guarantee that relevant information can be extracted, and that architectural changes can be propagated into the running system.

4. STYLE-BASED ADAPTATION

We now elaborate each aspect of this framework.

4.1 Architectural Models and Styles

The centerpiece of our approach is the use of stylized architectural models. Although there are many modeling languages and representation schemes for architecture, we adopt a simple approach in which an architectural model is represented as an annotated, hierarchical graph.[1] Nodes in the graph are *components*, which represent the principal computational elements and data stores of the system. Arcs are *connectors,* which represent the pathways of interaction between the components. Components and connectors have explicit interfaces (termed *ports* and *roles,* respectively). To support various levels of abstraction and encapsulation, we allow components and connectors to be defined by more detailed architectural descriptions, which we call *representations.*

To account for various semantic properties of the architecture we allow elements in the graph to be annotated with extensible property lists. Properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). Properties associated with a component might define its core functionality, performance attributes (e.g., average time to process a request, load, etc.) or reliability.

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 [9, 29, 30]. Moreover, the notion of style often maps

[1] This is the core architectural representation scheme adopted by a number of ADLs, including Acme [10], xArch [6], xADL [7], ADML [25], and SADL [22].

```
Family ClientServerFam = {
    Component Type ClientT = {...};
    Component Type ServerT = {...};

    Component Type ServerGroupT = {...};

    Role Type ClientRoleT = {...};

    Connector Type LinkT = {
        invariant size(select r : role in Self.Roles |
                declaresType(r, ServerRoleT)) == 1;
        invariant size(select r : role in Self.Roles |
                 declaresType(r, ClientRoleT)) >= 1;
        Role ClientRole1 : ClientRoleT;
        Role ServerRole : ServerRoleT;
    };
};
```

Figure 3. Client-Server Style Definition.

well to widely-used component integration infrastructures (such as EJB, HLA, CORBA), which prescribe the kinds of components allowed and the kinds of interactions that may take place between them.

As a result a number of ADLs and their toolsets have been created to support system development and execution for specific styles. For example, C2 [29] supports a style based on hierarchical publish subscribe; Wright [1, 2] supports a style based on formal specification of connector protocols; MetaH [30] supports a style based on real-time avionics control components.

In our research we adopt the view that while choice of style is critical to supporting system design, execution, and evolution, different styles will be appropriate for different systems. For example, a client-server system, such as the one in our example, will most naturally be represented using a client-server style. In contrast, a signal processing system would probably adopt a pipe-filter style. While one might *encode* these systems in some other style, the mapping to the actual system would become much more complex, with the attendant problems of making sure that any observation derived from the architecture has a bearing on the system itself.

```
Family PerformanceClientServerFam extends ClientServerFam with {
    Component Type PAClientT extends ClientT with {
        Properties  {
            Requests : sequence <any>;
            ResponseTime : float;
            ServiceTime : float;
        };
    };
    Connector Type PALinkT extends LinkT with {
        Properties {
            DelayTime : float;
        };
    };
    Component Type PAServerGroupT extends ServerGroupT with {
        Properties {
            Replication : int <<default : int = 1;>>;
            Requests : sequence <any>;
            ResponseTime : float;
            ServiceTime : float;
            AvgLoad : float;
        };
        Invariant AvgLoad > minLoad;
    };
    Role Type PAClientRoleT extends ClientRoleT with {
        Property averageLatency : float;
        Invariant averageLatency < maxLatency;
    };

    Property maxLatency : float;
    Property minLoad : float;
};
```

Figure 4. Client Server Style Extended for Performance Analysis.

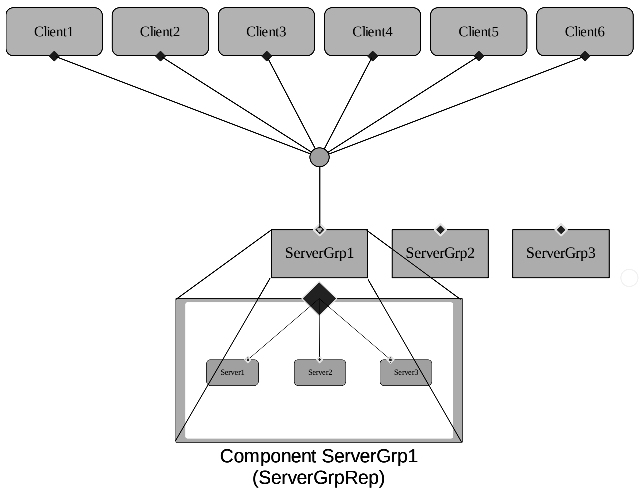

Figure 5. Architectural Model of the Example System.

For this reason, two key elements of our approach are the explicit definition of style and its accessibility at run time for system adaptation. Specifically, we define a style as a system of types, plus a set of rules and constraints. The types are defined in Acme [10], a generic ADL that extends the above structural core framework with the notion of style. The rules and constraints are defined in Armani [21] a first-order predicate logic similar to UML's OCL [24], augmented with a small set of architectural functions. These functions make it easier to define logical expressions that refer to things like connectedness, type conformance, and hierarchical relationships.[2] We say that a system *conforms* to a style if it satisfies all of the constraints defined by the style (including type conformance).

To illustrate, Figure 3 contains a partial description of the style used to characterize the class of web-based systems of our example. The style is actually defined in two steps. The first step specifies a generic client-server style (called a *family* in Acme). It defines a set of component types: a web client type (*ClientT*), a server group type (*ServerGroupT*), and a server (*ServerT*). It also defines a connector type (*LinkT*). Constraints on the style (appearing in the definition of *LinkT*) guarantee that the link has only one role for the server and more than one role for the client. Other constraints, not shown, define further structural rules (for example, each client must be connected to a server).

There are potentially many possible kinds of analysis that one might carry out on client-server systems built in this style. Since we are particularly concerned with overall system performance, we augment the client-server style to include performance-

[2] Details on Acme and Armani can be found elsewhere [10, 21]. Here we focus on how those representation schemes, originally developed as design-time notations, are extended and used to support run time adaptation.

oriented properties. These include the response time and degree of replication for servers and the delay time over links. This style extension is shown in Figure 4. Constraints on this style capture the desired performance related behavior of the system. The first constraint, associated with *PAServerGroupT,* specifies that a server group cannot be under-utilized. The second constraint, as part of the *PAClientRoleT,* specifies that the latency on this role cannot be above some specified maximum.

Having defined an appropriate style, we can now define a particular system configuration in that style, such as the one illustrated in Figure 5.

4.2 Style-specific Analytical Methods

As we argued above, one of the main benefits of style-based development is the ability to use analytical methods to evaluate properties of a system's architectural design. For example, MetaH uses real-time schedulability analysis, and Wright uses protocol model checking.

To illustrate how this works, consider our web style example. The use of buffered request queues, together with replicated servers, suggests the use of queuing theory as a basis for understanding the performance characteristics of systems built in this style. As we have shown elsewhere [28], for certain architectural styles queuing theory is useful for determining various architectural properties including system response time, server response time (*Ts*), average length of request queues (*Qs*), expected degree of server utilization (*Us*), and location of bottlenecks.

In the case of our example style, we have an ideal candidate for M/M/m analysis. The *M/M* indicates that the probability of a request arriving at component *s,* and the probability of component *s* finishing a request it is currently servicing, are assumed to be exponential distributions (also called "memoryless," independent of past events); requests are further assumed to be, at any point in time, either waiting in one component's queue, receiving service from one component, or traveling on one connector. The *m* indicates the replication of component *s*; that is, component *s* is not limited to representing a single server, but rather can represent a server group of *m* servers that are fed from a single queue. Given estimates for clients' request generation rates and servers' service times (the time that it takes to service one request), we can derive performance estimates for components according to Table 1. To calculate the expected system response time for a request, we must also estimate the average delay *Dc* imposed by each connector *c*, and calculate, for each component *s* and connector *c*, the average number of times (*Vs*, *Vc*) it is visited by that request. (Given *Vs* and the rates at which client components generate requests, we can derive rather than estimate *Rs,* the rate at which requests arrive at server group *s*.)

Applying this M/M/m theory to our style tells us that with respect to the average latency for servicing client requests, the key design parameters in our style are (a) the replication factor *m* of servers within a server group, (b) the communication delay *D* between clients and servers, (c) the arrival rate *R* of client requests and (d) the service time *S* of servers within a server group.

In previous work [28] we showed how to use that analysis to provide an initial configuration of the system based on estimates of these four parameters. In particular, Equation (5) in Table 1 indicates for each server group a design tradeoff between utilization (underutilized servers waste resource) and response time. Utilization is in turn affected by service time and replication. Thus, given a range of acceptable utilization and response time, if we choose service time then replication is constrained to some range (or vice versa). As we will show in the next section, we can also use this observation to determine sound adaptation policies.

We can use the performance analysis to decide the following questions about our architecture, assuming that the requirements for the initial system configuration are that for six clients each client must receive a latency not exceeding 2 seconds for each request and a server group must have a utilization of between 70% and 80%:

- How many replicated servers must exist in a server group so that the server group is properly utilized?
- Where should the server group be placed so that the bandwidth (modeled as the delay in a connector) leads to latency not exceeding 2 seconds?

Given a particular service time and arrival rate, performance analysis of this model gives a range of possible values for server utilization, replication, latencies, and system response time. We can use Equation (5) to give us an initial replication count and Equation (6) to give us a lower bound on the bandwidth. If we assume that the arrival rate is 180 requests/sec, the server response time is between 10ms and 20ms the average request size is 0.5KB, and the average response size is 20KB, then the performance analysis gives us the following bounds:

Initial server replication coun t= 3-5
Zero delay System Response Time = 0.013-0.026 seconds

Therefore,

0 < Round-trip connector delay < 1.972 seconds, or
0 < Average connector delay < .986 seconds

Thus, the average bandwidth over the connector must be greater than 10.4KB/sec.

4.3 Using Styles to Assist Adaptation

The representation schemes for architectures and style outlined above were originally created to support design-time development tools. In this section we show how styles can be augmented to function as run time adaptation mechanisms. We then consider the

Table 1. Performance Equations From [4]

(1)	Utilization of server group *s*	$u_s = \frac{R_s S_s}{m}$
(2)	Probability {no servers busy}	$p_0 = \left[\sum_{i=0}^{m} \frac{(mu_s)^i}{i!} + \frac{u_s(mu_s)^m}{m!(1-u_s)}\right]^{-1}$
(3)	Probability {all servers busy}	$P_Q = \frac{p_0(mu_s)^m}{m!(1-u_s)}$
(4)	Average queue length of *s*	$Q_s = \frac{P_Q u_s}{1-u_s}$
(5)	Average response time of *s*	$T_s = S_s + \frac{P_Q u_s}{R_s(1-u_s)} =$ $S_s + \frac{S_s(mu_s)^m}{mm!(1-u_s)^2 \sum_{n=0}^{m} \frac{(mu_s)^n}{n!} + (1-u_s)(mu_s)^{m+1}}$
(6)	System response time (latency)	$\sum T_s V_s + \sum D_c V_c$

supporting run time infrastructure needed to make this work out in practice in Section 4.4.

Two key augmentations to style definitions are needed to make them useful for run time adaptation: (1) the definition of a set of adaptation operators for the style, and (2) the definition of a set of repair strategies.

4.3.1 Adaptation operators

The first extension is to augment a style description with a set of operators that define the ways one can change instances of systems in that style. Such operators determine a "virtual machine" that can be used at run time to adapt an architectural design.

Given a particular architectural style, there will typically be a set of natural operators for changing an architectural configuration and querying for additional information. In the most generic case, architectures can provide primitive operators for adding and removing components and connections [26]. However, specific styles can often provide much higher-level operators that exploit the restrictions in that style and the intended implementation base.

Two key factors determine the choice of operators for a style. First is the style itself – the kinds of components, connectors and configuration rules. Based on its constraints, a style can both limit the set of operations, and also suggest a set of higher-level operators. For example, if a style specifies that there must be exactly one instance of a particular type of component, such as a database, the style should not provide operations to add or remove an existing instance of this type. On the other hand, if another constraint says that every client component in the system must be attached to the (unique) database, it would make sense that a "new-client" operation would automatically create a new client-database connector and attach it between the new component and the database. These style-specific operators are defined in terms of style-neutral operators such as "add a component" or "remove a connector." The definition of these style-neutral operations can be based on [31] or [32].

The second factor is the feasibility of carrying out the change. To evaluate feasibility requires some knowledge of the target implementation infrastructure. It makes no sense to prescribe an architectural operator that has no hope of ever being carried out on the running system. For some styles, the relation is defined by construction (since implementations are generated from architectures). More generally, however, the style designer may have to make certain assumptions about the availability of implementation-changing operators that will be provided by the run time environment of the system. (We return to this issue in Section 6.)

In terms of our example, we define the following operators:

- **addServer**(): This operation is applied to a component of type *ServerGroupT* and adds a new component of type *ServerT* to its representation, ensuring that there is a binding between its port and the ServerGroup's port.
- **move**(to:ServerGroupT): This operation is applied to a client and deletes the role currently connecting the client to the connector that connects it to a server group and performs the necessary attachment to a connector that will connect it to the server group passed in as a parameter.
- **remove**(): This operation is applied to a server and deletes the server from its containing server group. Furthermore, it changes the replication count on the server group and deletes the binding.

The above operations all effect changes to the model. The next operation queries the state of the running system:

- **findGoodSGroup**(cl:ClientT,bw:float):ServerGroupT; finds the server group with the best bandwidth (above *bw*) to the client *cli*, and returns a reference to the server group.

These operators reflect the considerations just outlined. First, from the nature of a server group, we get the operations of activating or deactivating a server within a group. Also, from the nature of the asynchronous request connectors, we get the operations of adapting the communication path between particular clients and server groups. Second, based on the knowledge of supported system change operations, outlined in Section 4.4, we have some confidence that the architectural operations are actually achievable in the executing system.

4.3.2 Repair strategies

The second extension to the notion of style is the specification of repair strategies that correspond to selected constraints of the style. The key idea is that when a stylistic constraint violation is detected, the appropriate repair strategy will be triggered.

Describing Strategies

A repair strategy has two main functions: first to determine the cause of the problem, and second to determine how to fix it. Thus the general form of a repair strategy is a sequence of repair *tactics*. Each repair tactic is guarded by a pre-condition that determines whether that tactic is applicable. The evaluation of a tactic's precondition will usually involve the examination of various properties of the architecture in order to pinpoint the problem and determine applicability. If it is applicable, the tactic executes a repair script that is written as an imperative program using the style-specific operators described above.

To handle the situation that several tactics may be applicable, the enclosing repair strategy decides on the policy for executing repair tactics. It might apply the first tactic that succeeds. Alternatively, it might sequence through all of the tactics, or use some other style-specific policy.

The final complication associated with repair strategies is the use of transactions. The body of a repair strategy is typically enclosed within a transactional scope so that if an error occurs during the execution of a repair, the system can abort the repair, leaving the architecture in a consistent state. Failure of a repair strategy can be caused by a number of factors. For example, it may be the case that none of the tactics have applicable firing conditions. Or, an applicable tactic may find that conditions of the actual system or its environment do not permit it to carry out its repair script. Transaction aborts cause the system to inform the user of a system error that cannot be handled by the automated mechanisms.

```
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 6. Repair Strategy for High Latency.

Choosing Tactics

One of the principal advantages of allowing the system designer to pick an appropriate style is the ability to exploit style-specific analyses to determine whether repair tactics are sound. By sound, we mean that if executed the changes will help reestablish the violated constraint.

In general an analytical method for an architecture will provide a compositional method for calculating some system property in terms of the properties of its parts. For example, a reliability analysis will depend on the reliability of the architectural parts; a performance analysis will depend on various performance attributes of the parts. By looking at the constraint to be satisfied, the analysis can often point the repair strategy writer both to the set of possible causes for constraint violation, and for each possible cause, to an appropriate repair.

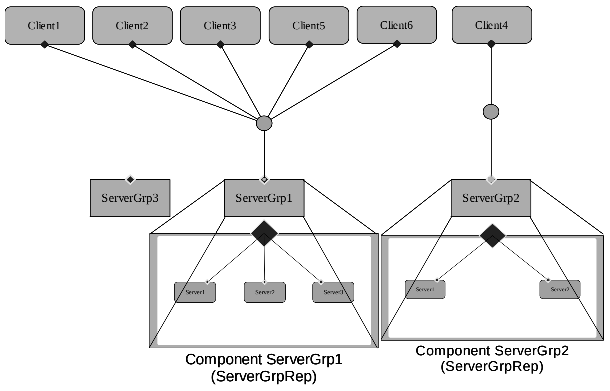

Figure 7. Model of System after Low Bandwidth Repair

Illustrating this idea for our example, we can show how the repair strategy developed from the theoretical analysis. The equations for calculating latency for a service request (Table 1) indicate that there are four contributing factors: 1) the connector delay, 2) the server replication count, 3) the average client request rate, and 4) the average server service time. Of these we have control over the first two. When the latency is high, we can decrease the connector delay or increase the server replication count to decrease the latency. Determining which tactic depends on whether the connector has a low bandwidth (inversely proportional to connector delay) or if the server group is heavily loaded (inversely proportional to replication). These two system properties form the preconditions to the tactics; we have thus developed a repair strategy with two tactics.

Applying Our Approach

Figure 6 (line 1-3) illustrates the repair strategy associated with the latency threshold constraint. In line 2, "!→" denotes "if constraint violated, then execute." The top-level repair strategy in lines 5-17, *fixLatency*, consists of two tactics. 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. The result of this repair on Figure 5 is depicted in Figure 7. The repair strategy uses a policy in which it executes these two tactics sequentially: if the first tactic succeeds it commits the repair strategy; otherwise it executes the second. The strategy will abort if neither tactic succeeds, or if the second tactic finds that it cannot proceed since there are no suitable server groups to move the connection to.

4.4 Bridging the Gap to Implementation

As we have argued above, the use of style allows us to provide automated support for architectural adaptation *at the model level.* That is, we can use the constraints, operators, and analytical methods to determine how to modify the architecture.

The only catch is that we somehow have to relate all of that to the real world. There are two parts to this. The first is getting information out of the executing system so we can determine when architectural constraints are violated. The second is propagating architectural repairs into the system itself.

4.4.1 Monitoring

In order to provide a bridge from system level behavior to architecturally-relevant observations, we have defined a three-level approach illustrated in Figure 8. This monitoring infrastructure is described in more detail elsewhere [11]: here we summarize the main features, stressing the connection with style specifications.

The lowest level is a set of *probes*, which are "deployed" in the target system or physical environment. Probes monitor the system and announce observations via a "probe bus." At the second level a set of *gauges* consume and interpret lower-level probe measurements in terms of higher-level model properties. Like probes, gauges disseminate information via a "gauge reporting bus." The top-level entities in Figure 8 are *gauge consumers*, which consume information disseminated by gauges. Such information can be used, for example, to update an abstraction/model, to make system repair decisions, to display warnings and alerts to system users, or to show the current status of the running system.

The separation of the monitoring infrastructure into these parts helps isolate separable concerns. Probes are highly implementation-specific, and typically require detailed knowledge of the execution environment. Gauges are model-specific. They need only understand how to convert low-level observations into properties of more abstract representations, such as architectural models. Finally, gauge consumers are free to use the interpreted information to cause various actions to occur, such as displaying warnings to the user or automatically carrying out repairs.

In the context of architectural repair, we use the architectural style to inform us where to place gauges. Specifically, for each constraint that we wish to monitor, we must place gauges that dynamically update the properties over which the constraint is defined. In addition, our repair strategies may require additional monitored information to pinpoint sources of problems and execute repair operations.

For instance, in the example above we are concerned with the average latency of client requests. To monitor this property, we must associate a gauge with the *averageLatency* property of each client role (see the definition of *ClientRoleT* in Figure 4). This latency gauge in turn deploys a probe into the implementation that monitors the timing of reply-request pairs. When it receives such monitored values it averages them over some window, updating the latency property in the architecture model when it changes.

But average latency is not the only architectural property that we need to monitor. The repair tactics, derived from queuing theoretic model of performance analysis, rely on information about two additional constraints: the bandwidth between the client and the server is low or the server group is overloaded (or both). Thus, to determine why latency is high in the architecture, we need to monitor these two properties. The gauge for measuring bandwidth uses the same probe used by the latency gauge for measuring the time it takes to receive a reply. An additional probe measures the size of the reply and calculates the bandwidth based on these values. Determining the load on the server can be done in a number of ways. For example, we could take the loads of the machines running each server and average these within a group to determine the load; alternatively, we could measure the size of a request queue to indicate whether the server group is overloaded. In our current implementation we adopt the latter approach.

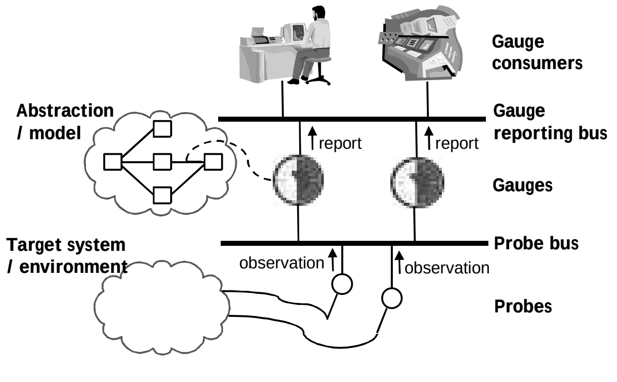

Figure 8. Gauge Infrastructure

Table 2. Environment Manager Operators and Queries.

createReqQueue()	Adds a logical request queue to *Req-queue* machine in Figure 2.
findServer([*string cli_ip, float bw_thresh*])	Finds a spare server that has at least *bw_thresh* bandwidth between it and the client.
moveClient(ReqQ newQ)	Moves a client to the new request queue.
connectServer(Server srv, ReqQ to)	Configures a server so that it pulls client requests out of the *to* request queue.
activateServer ()	Signals that the server should begin pull requests from the request queue.
deactivateServer()	Signals that a server should stop pulling requests from the request queue.
remos_get_flow (string clIP, string svIP)	This is a Remos API call that returns the predicted bandwidth between two IP addresses.

4.4.2 Repair execution

The final component of our adaptation framework is a translator that interprets repair scripts as operations on the actual system (Figure 1, item 6). As we noted earlier, we assume that the executing system provides a set of system-changing operations via an Environment Manager. The nature of these operations will depend heavily on the implementation platform. In general, a given architectural operation will be realized by some number of lower level system reconfiguration operations. Each such operator can raise exceptions to signal a failure. The Translator then propagates them to the model level, where transactions boundaries can cause the repair strategy to abort.

To illustrate, the specific operators and queries supported by the Environment Manager in our example are listed in Table 2. These operators include low-level routines for creating new request queues, activating and deactivating servers, and moving client communications to a new queue

The Translator for our example maps the Model API operations described in Section 4.3.1 to the Environment Manager operations using the scheme summarized in Table 3. (Parameters passed between the levels also need to be translated. We do not discuss this here.) The actual map involves mapping model-level parameters to implementation level parameters, and mapping return values to model values.

4.4.3 Putting the Pieces Together

Let us summarize how the parts work together, end-to-end, and how pieces of the framework in Figure 1 interact. While the system is running, relevant system properties are observed and collected by gauges in the Monitoring Mechanisms and updated on the Architectural Model. Whenever there is a change in a gauge value, the Constraint Evaluator in the Architecture Manager re-evaluates the architectural constraints to check for violation. Suppose that a latency constraint violation is detected in some Client role, then the Constraint Evaluator calls the Repair Handler to trigger a repair. The Repair Handler first signals the Constraint Evaluator to suspend all monitoring and captures a "snapshot" of the current state of the Architectural Model – doing so prevents other constraint violation from interfering with the present repair and preserves the property values at the time of constraint violation to facilitate decision-making. The Repair Handler then begins running the repair script, *fixLatency*. The first statement (line 6 of Figure 6) in *fixLatency* signals the Transaction Handler to begin a repair transaction.

During the repair, the Transaction Handler keeps track of all the Model API calls and listens for any abort signals. If an abort signal is raised either by a Model API operation or the Repair Handler while running the script, the Transaction Handler rolls back any changes made to the Architectural Model, and the error is propagated to a human operator. If no abort occurs, the repair script completes with a commit operation, and the Transaction Handler feeds the collected Model API operations to the Translator for translation to Environment API calls.

The Translator hands the translated script to the Environment Manager, which executes it to make changes to the Executing System. The Environment operations have exceptions not shown which may be raised if execution fails. The Translator would then pass the exception signal back to the Transaction Handler, which aborts the repair transaction. However, if the Environment Manager returns a successful status, then the Transaction Handler commits the repair changes on the model. Whether the repair transaction commits or aborts, the Repair Handler signals to the Constraint Evaluator to resume system monitoring and resets appropriate gauges.

At this point, as part of the dynamic verification to ensure that the repair was effective, the constraints are re-evaluated to determine whether any violations are now fixed, and the repair cycle completes. If a violation remains, or if a new violation is detected, the repair is triggered again and the process repeats.

5. IMPLEMENTATION STATUS

In terms of the adaptation framework in Figure 1, our implementation contains the following pieces:

Monitoring Mechanisms: Our approach is general enough to be used with existing technologies for monitoring systems and their environments. To connect with the infrastructure described in this paper, a wrapper needs to be written for these technologies that allows events to be generated according to the probe infrastructure, mentioned in Figure 8, turning the technology into a probe. We have developed prototype probes for gathering information about networks, based on the Remos system [17]. Remos has two parts: 1) an API, that allows applications to issue queries about bandwidth and latency between groups of hosts, implemented as a library that is linked with the application; and 2) a set of servers, called collectors, that collect information about different parts of the network [20]. A probe uses Remos to collect the information required for the probe and distributes it as events using the Siena wide area event bus [5]; gauges listen to this information and perform calculations and transformations to relate it to the software architecture of the system.

We are currently developing a number of gauges that can be used in various architectural styles. These gauges range from simple averaging gauges, that average information collected from probes, to more complex protocol matching gauges, that watch events occurring in the implementation and match them to a statechart specification of their protocol expressed in the architecture. Others are also developing gauges to fit with the infrastructure described in this paper.

Architectural Models: AcmeStudio, a design environment that allows architectures to be described in Acme, has been modified so that it provides an architectural model at run time. It receives monitored information at run time, and also allows the placement of gauges in the architecture.

Constraint Evaluator: We have changed our tool for evaluating Armani constraints at design time so that it evaluates constraints dynamically.

Repair Handler: The Armani constraint handler has been augmented so that it supports the specification and execution of repairs.

Translator and **Environment Manager**: Currently, we have hand-tailored support for these components that need to be changed for each implementation. Our work in this area will concentrate on providing more general mechanisms where appropriate, and perhaps using off-the-shelf reconfiguration commands for commercial systems.

6. DISCUSSION

We have described an approach in which architectural-based self-adaptation is supported by the incorporation of styles as an explicit design choice in the adaptation framework. The flexibility inherent in this approach permits the system maintainer to pick a style that matches well to existing implementation bases, provides a formal basis for specifying constraints, and can permit the definition of repair policies that are justified by analytic methods.

However, this flexibility also introduces several new complexities over other approaches in which the choice of architectural style is hardwired into the framework. In particular, at least three critical questions are raised: First, is it always possible to map architectural repairs into corresponding system changes?

Table 3. Mapping Between Model and Implementation Operators.

Model Level	Environment Level
addServer	**findServer** **activateServer** **connectServer**
moveClient	**createReqQue** **moveClient**
findGoodSGrp	Conditionals + multiple calls to **remos_get_flow**

Second, is it always possible to monitor relevant run time information? Third, is it reasonable to expect that analytical techniques can address a sufficiently broad set of concerns to inform our repair strategies? We address each issue in turn.

Model-Implementation Map: In our approach the ability to map architectural changes to corresponding implementation reconfigurations is moderated by two factors. First is an assumption that systems provide a well-defined set of operations for modifying a running system. Of course, in general this may not be true. Some systems are inherently unreconfigurable, in which case our approach would simply not work. However, many systems do in fact embody changing operations – such as the ability to load dynamic libraries and remote code, to redirect communications to alternative pathways, or to do dynamic resource management. Moreover, we would argue that such capabilities are going to be increasingly prevalent in modern systems that are intended to function in a connected, web-based universe. For example, modern frameworks like Jini provide as a fundamental building block the notion of allocation and deallocation of resources, and location-independence of services.

The other moderating factor is an assumption that architectural style is not chosen arbitrarily. Obviously, attempting to pair an arbitrary style with an arbitrary implementation could lead to considerable difficulty in relating the two. However, one of the hallmarks of our approach is that it encourages one to match an appropriate style to an implementation base. Hence, in fact, the flexibility of choosing a style can actually help reduce the distance between system implementations and architectural models.

Implementation-Model Map: For our approach to work it must be possible to reflect dynamic system state into an architectural model. To do this we provide a multi-leveled framework that separates concerns of low-level system instrumentation from concerns of abstracting those results in architecturally meaningful terms. What makes us think that either part will be feasible in the general case?

The ability to monitor systems is itself an active research area. Increasingly systems are expected to provide information that can be used to determine their health. Moreover, there are an increasingly large number of non-intrusive post-deployment monitoring schemes. For example, to deal with network performance we were able to use a monitoring infrastructure developed completely independently. It in turn relies on standard protocols for SNMP. Other researchers and practitioners are developing many other schemes such as the ability to place monitors between COM components, the ability to monitor network traffic to determine security breaches, the ability to monitor object method calls, and various probes that determine whether a given component is alive.

In terms of mapping low-level information to architectural information, the capability will certainly depend on the distance between the architectural and implementation styles. As we argued earlier, our approach encourages developers to pick styles where that mapping will be straightforward.

Analytical Methods: A key feature of our approach is the notion that repair strategies should leverage architectural analyses. We demonstrated one such analysis for performance. What makes us think that others exist?

In fact, there is considerable work recently on finding good architecture-based analyses. For example, Klein et al. [16] provide a method of reasoning about the behavior of component types that interact in a defined pattern. In earlier work we showed how to adapt protocol analysis to architectural modification [2]. Others have shown how real-time schedulability can be applied [30]. Although far from providing a complete repertoire of analytical techniques, the space is rich, and getting richer.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a technique for using software architectural styles to automate dynamic repair of systems. In particular, styles and their associated analyses

- make explicit the constraints that must be maintained in the face of evolution
- direct us to the set of properties that must be monitored to achieve system quality attributes and maintain constraints
- define a set of abstract architectural operators for repairing a system
- allow us to select appropriate repair strategies, based on analytical methods

We illustrated how the technique can be applied to performance-oriented adaptation of certain web-based systems.

For future research we see opportunities to improve each of the areas mentioned in Section 6. We need to be able to develop mechanisms that provide richer adaptability for executing systems. We need new monitoring capabilities, and reusable infrastructure for relating monitored values to architectures. We need new analytical methods for architecture that will permit the specification of principled adaptation policies.

Additionally we see a number of other key future research areas. First is the investigation of more intelligent repair policy mechanisms. For example, one might like a system to dynamically adjust its repair tactic selection policy so that it takes into consideration the history of tactic effectiveness: effective tactics would be favored over those that sometimes fail to produce system improvements. Second is the link between architectures and requirements. Systems may need to adapt, not just because the underlying computation base changes, but because user needs change. This will require ways to link user expectations to architectural parameters and constraints. Third is the development of concrete instances of our approach for some of the common architectural frameworks, such as EJB, Jini, and CORBA.

8. ACKNOWLEDGMENTS

DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616, supports this work. 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.

9. 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] Allen, R.J., Douence, R., and Garlan, D. Specifying Dynamism in Software Architectures. In Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997.

[3] Allen, R.J and Garlan, D. A Formal Basis for Architectural Connection. ACM Transactions of Software Engineering and Methodology, July 1997.

[4] Bertsekas, D. and Gallager, R. Data Networks, Second Edition. Prentice Hall, 1992. ISBN 0-13-200916-1.

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

[6] Dashofy, E., Garlan, D., van der Hoek, A., and Schmerl, B. http://www.ics.uci.edu/pub/arch/xarch/.

[7] Dashofy, E., van der Hoek, A., and Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 2001.

[8] Gantenbien, R.E. Dynamic Binding in Strongly Typed Programming Languages. *Journal of Systems and Software* **14**(1):31-38, 1991.

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

[10] 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.

[11] Garlan, D., Schmerl, B.R., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. The Working Conference on Complex and Dynamic System Architecture. Brisbane, Australia, December 2001. To appear.

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

[13] Gorlick, M.M. Distributed Debugging on $5 a day. Proceedings of the California Software Symposium, University of California, Irvine, CA, 1997 pp. 31-39.

[14] Gosling, J. and McGilton, H. The Java Language Environment: A White Paper. Sun Microsystems Computer Company, Mountain View, California, May 1996. Available at http://java.sun.com/docs/white/langenv/.

[15] Ho, W.W. and Olsson, R.A. An Approach to Genuine Dynamic Linking. *Software – Practice and Experience* **21**(4):375—390, 1991.

[16] Klein, M., Kazman, R., Bass, L., Carriere, J., Barbacci, M., Lipson, H. Attribute-Based Architecture Styles. Software Architecture (Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)), (San Antonio, TX), February 1999, 225-243.

[17] 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.

[18] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. Proceedings of 5th European Software Engineering Conference (ESEC '95), Sitges, September 1995. Also published as Lecture Notes in Computer Science 989, (Springer-Verlag), 1995, pp. 137-153.

[19] Métayer, D.L. Describing Software Archtiecture Styles using Graph Grammars. *IEEE Transactions on Software Engineering*, **24**(7):521-553, July 1998.

[20] Miller, N., and Steenkiste, P. Collecting Network Status Information for Network-Aware Applications. IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.

[21] Monroe, R.T. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.

[22] 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.

[23] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S., and Kirby, G.N.C. Exploiting Persistent Linkage in Software Engineering Environments. *The Computer Journal* **38**(1):1—16, 1995.

[24] Object Management Group. The OMG Unified Modeling Language Specification, Version 1.4. September 2001. Available at http://www.omg.org/technology/documents/formal/uml.htm.

[25] The OpenGroup. Architecture Description Markup Language (ADML) Version 1. April, 2000. Available at http://www.opengroup.org/publications/catalog/i901.htm.

[26] Oriezy, P., Medvidovic, N., and Taylor, R.N. Architecture-Based Runtime Software Evolution in the Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, April 1998, pp. 11—15.

[27] 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.

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

[29] 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.

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

[31] Wermelinger, M., Lopes, A., and Fiadeiro, J.L. A Graph Based Architectural (Re)configuration Language. Proceedings of the Joint 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Vienna, Austria, September 2001, pp. 21—32.

[32] Wile, D.S. AML: An Architecture Meta-Language. In Proceedings of the Automated Software Engineering Conference, Cocoa Beach, FL, October 1999.