In Software Architecture: System Design, Development, and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture) Jan Bosch, Morven Gentleman, Christine Hofmeister, Juha Kuusela (Eds), Kluwer Academic Publishers, August 25-31, 2002. pp. 45-59.

Using Architectural Style as a Basis for System Self-repair

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
{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. Recently several researchers have explored the possibility of using architectural models as a basis for run time monitoring, error detection, and repair. Each of these efforts, however, has demonstrated the feasibility of using architectural models in the context of a specific style. In this paper we show how to generalize these solutions by making architectural style a parameter in the monitoring/repair framework and its supporting infrastructure. The value of this generalization is that it allows one to tailor monitoring/repair mechanisms to match both the properties of interest (such as performance or security), and the available operators for run time adaptation.

Key words: 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.).

Recently a number of researchers have proposed an 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 [21] rather than relying on system-specific built-in mechanisms. 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 architecture description languages (ADLs) and toolsets) to provide intrinsically modifiable architectures, or the use of low-level architectural adaptation operations. Taylor and colleagues use hierarchical publish-subscribe via C2 [20, 23]; Gorlick and colleagues use data-flow style via Weaves [10]; Magee and colleagues use bi-directional communication links via Darwin [14]; and Wermelinger and colleagues [25] use architectural primitives, independent of particular architectural styles, to effect architectural changes.

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 or views 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; and (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 behaviour, 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 behaviour. 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., [6, 12, 13, 18]). Systems of this kind allow self-repair to be programmed on a per-system basis. For self-repair to be useful in a large range of mobile, defence, or e-commerce systems, it is desirable that application writers not code specific solutions for each application; nor should users of each application be burdened with interacting with different change mechanisms. Rather, we require external, reusable mechanisms that can be added to systems in a disciplined manner.

A more disciplined approach can be found in the area of distributed debugging systems [11]. 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 must introduce some new mechanisms to allow “run time” styles to be treated as design parameters 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 [10, 23]) this is less of an issue because architectures are closely coupled to their implementations by construction.

Finally, there has been some work on formally characterizing architecture styles, and using them as a basis for static system analysis [7, 22]. Our research extends this by showing how to turn “style as a design time artefact” into “style as a run time artefact”. 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, 15, 25].

3. OVERVIEW OF APPROACH

Our starting point is an architecture-based approach to self-adaptation, similar to [21] (as illustrated in Figure 1): An executing system (1) is monitored to observe its run time behaviour (2). Monitored values are abstracted and related to architectural

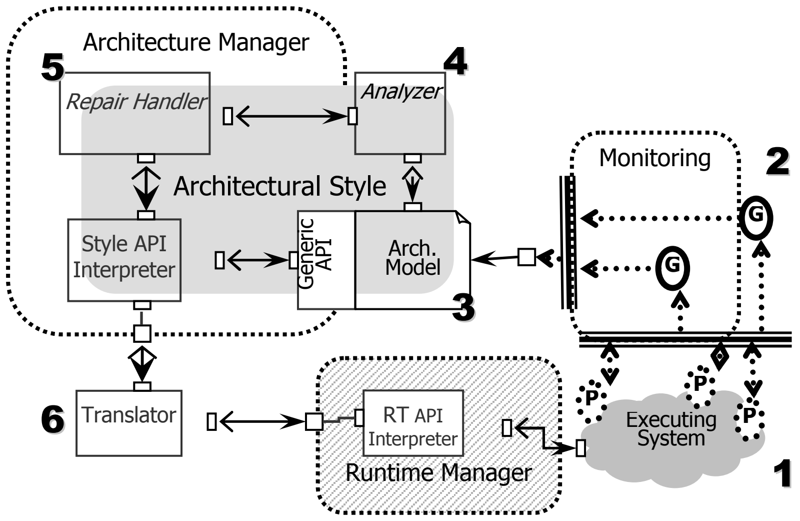

Figure 1. Adaptation Framework.

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 behaviour 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 introduce a style-specific translation component to 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 interrelated 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 reduce costs, the 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.

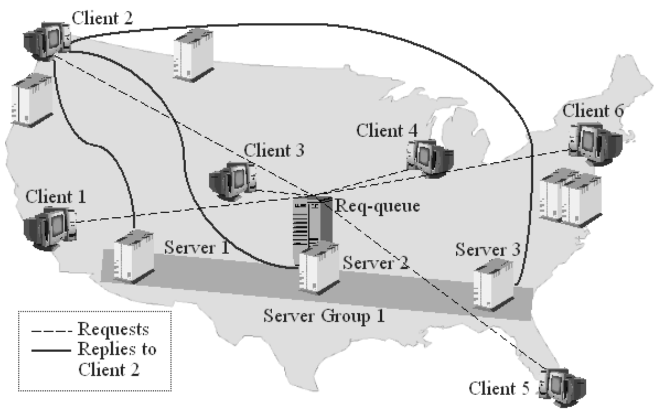

Figure 2. Deployment Architecture of the Example System.

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 (Figure 3(a)). Next, because we are focussing on performance, we adapt that style so that it exposes performance related properties and makes explicit constraints about performance (Figure 3(b)). 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 artefact 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." To 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 re-evaluation 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, focussing on the way stylized architectural models support problem detection and repair. Given limits of space, we omit details on the monitoring and run time system change infrastructure, which are described elsewhere [4, 9].

4.1 Architectural Models and Styles

The centrepiece of our approach is the use of stylized architectural models. We adopt an 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, elements in the graph can 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, etc.

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 an *architectural 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.

[1] This is, in fact, the core architectural representation scheme adopted by a number of ADLs, including Acme [8], xADL [5], and SADL [17].

Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [7, 23, 24]. Moreover, the notion of style often maps 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 [23] supports a style based on hierarchical publish subscribe; Wright [1, 2] supports a style based on formal specification of connector protocols; MetaH [24] 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.

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 [8], a generic ADL that extends the above structural core framework with the notion of style. The rules and constraints are defined in Armani [16], a first-order predicate logic similar to UML's OCL [19], 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. 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(a) 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 many possible kinds of analysis that one might carry out on client-server systems built in this style. Since we are concerned with overall system performance in this example, we augment the client-server style to include performance-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 3(b). Constraints on this style capture the desired performance related behaviour of the system. The first constraint, associated with *PAServerGroupT*, specifies that a server group cannot be under-utilized. The second constraint, as part of *PAClientRoleT*, indicates that the latency on this role should be below 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 4.

```
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;
    };
};
```

(a)

```
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;
};
```

(b)

Figure 3. (a) Client-Server Style Definition in Acme; (b) Client- Server Style Extended for Performance Analysis.

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, systems in the MetaH style use real-time schedulability analysis, while those in Wright can use protocol model checking [1, 2, 24].

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 under-standing the performance characteristics of systems built in this style. As we have shown elsewhere [22], for certain architectural styles queuing theory is useful for determining various architectural properties including system response time, server response time (T_s), average length of request queues (Q_s), expected degree of server utilization (U_s), and location of bottlenecks.

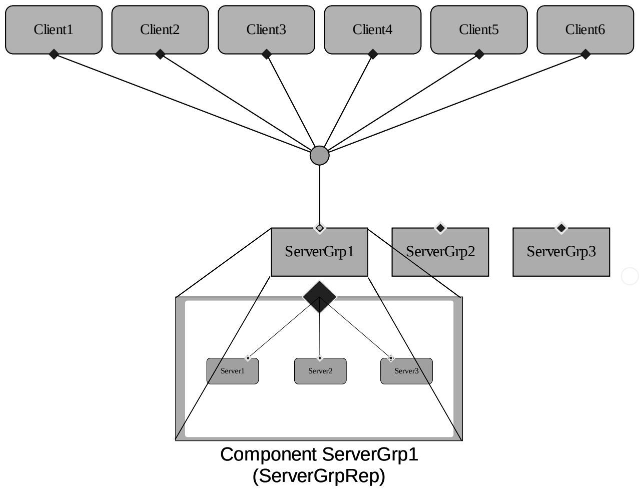

Figure 4. Architectural Model of the Example System.

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 travelling 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 D_c imposed by each connector *c*, and calculate, for each component *s* and connector *c*, the average number of times (V_s, V_c) it is visited by that request. (Given V_s and the rates at which client components generate requests, we can derive rather than estimate R_s, the rate at which requests arrive at server group *S*.)

Applying 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 [22] 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 trade-off between utilization (underutilized servers waste resources) 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 show in the Section 4.3, we can also use this observation to determine sound adaptation policies.

Table 1. Performance Equations From [3].

(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)	$\sum T_s V_s + \sum D_c V_c$

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 (modelled 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 count= 3-5 (in one server group)
Zero delay System Response Time = 0.013-0.026 seconds
0 < Round-trip connector delay < 1.972 seconds, or
0 < Average connector delay < .986 seconds
Average Bandwidth > 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. 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 model.

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 [20]. However, specific styles can often provide higher-level operators that exploit the restrictions of 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 prohibit addition or removal of an 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, 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 lower-level style-neutral operators such as "add component" or "remove connector," such as those defined in [25] or [26].

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

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 for 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.

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. When a stylistic constraint violation is detected, the appropriate repair strategy will be triggered.

4.3.2.1 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 precondition 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, such as the approach we use for this example. Alternatively, it might sequence through all of the tactics, or use some other style-specific policy.

4.3.2.2 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 re-establish 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 point the repair strategy writer both to the set of possible causes for constraint violation, and for each possible cause, to an appropriate repair.

Illustrating this idea in our example, the repair strategy developed from the theoretical performance analysis in the following way: 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, by finding another server group that has a higher bandwidth to a client, or increase the server replication count, by adding another server to a server group, 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 pre-conditions to the tactics; we have thus developed a repair strategy with two tactics.

```
01 invariant r.Avg_Latency <= maxLatency
02 !→
03     fixLatency(r);
04
05 strategy fixLatency (badRole: ClientRoleT) = {
06
07     let badClient: ClienT =
08         select one cli: ClientT in self.Components |
09             exists p: RequestT in cli.Ports |
10                 attached(badRole, p);
11     if (fixServerLoad(badClient)) {
12         commit;
13     else if (fixBandwidth(badClient, badRole) {
14         commit;
15     } else {
16         abort ModelError;
17     }
18 }
19
20 tactic fixServerLoad (client: ClientT) : boolean = {
21     let overloadedServerGroups: Set{ServerGroupT} =
22         {select sgp: ServerGroupT in self.Components |
23             connected(sgp, client) and
24             sgrp.Server_Load > maxServerLoad };
25     if (size(overloadedServerGroups) == 0) {
27     }
28     foreach sGrp in overloadedServerGroups {
29         sGrp.addServer();
30     }
31     return (size(overloadedServerGroups) > 0);
32 }
33
34     tactic fixBandwidth (client: ClientT,
35                             role: ClientRoleT) : boolean = {
36     if (role.Bandwidth >= minBandwidth) {
37         return false;
38     }
39     let oldSGrp: ServerGroupT =
40         select one sGrp: ServerGroupT in
41             self.Components |
42                 connected(client, sGrp);
43     let goodSGrp: ServerGroupT =
44         findGoodSGrp(client, minBandwidth);
45     if (goodSGrp != nil) {
46         client.moveClient(oldSGrp, goodSGrp);
47         return true;
48     } else {
49         abort NoServerGroupFound;
50     }
}
```

Figure 5. Repair Strategy for High Latency

4.3.2.3 Applying Our Approach

Figure 5 (line 1-3) illustrates the repair strategy associated with the latency threshold constraint. In line 2, "!→" denotes "execute on constraint violation." The top-level repair strategy, *fixLatency*, in lines 5-17, consists of two tactics. The first tactic in lines 20-32 handles the situation in which a server group is overloaded, identified by the precondition in lines 25-27. Its main action in lines 28-30 is to create a new server in any of the overloaded server groups. The second tactic in lines 34-51 handles the situation in which high latency is due to communication delay, identified by the precondition in lines 36-38. In lines 43-44, it queries the architecture to find a server group that will yield a higher bandwidth connection. In lines 45-47, if such a group exists it moves the client-server connector to use the new group. 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.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a technique for generalizing the use of arch models to support dynamic repair of systems. Extending earlier work by others, which demonstrated the value of architecture-based adaptation for specific styles of architecture, we have shown how to make the choice of style a parameter of the overall adaptation framework. The explicit incorporation of styles and their associated analyses allow one to

– make explicit the constraints that must be maintained in the face of evolution
– 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 client-server systems, and future work will involve applying our approach to the performance of commercial web-based systems, as well as other styles.

The components required in our approach to monitor the system and map architectural changes into run time system changes are discussed in [4, 9]. Briefly, the analysis associated with a style points us to properties in the architecture that need to be monitored dynamically. Gauges are attached to these properties and generate new property values based on information from probes that are deployed in the system implementation. Constraints associated with architectural properties are evaluated when the properties change to fire the repair strategies. We assume that there are some primitive, system-specific change operators into which we can map style-specific change operators. The system-specific operators may be as primitive as operating system calls to stop and start processes, or the system may provide its own change language that can be used in our framework.

For future research we intend to develop mechanisms that provide richer adaptability for executing systems. 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 favoured 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 also because user needs change. This will require ways to link user expectations to architectural parameters and constraints. Third is to apply our approach to some common architectural frameworks and styles, such as EJB, Jini, and CORBA. Fourth is to develop a more general analytic basis for determining whether a given repair strategy is *sound* (satisfies the constraints embodied in the style) and *stable* (converges to an improved state).

ACKNOWLEDGEMENTS

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.

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 Proc. the Workshop on Foundations of Component-Based Software Engineering, September 1997.

[3] Bertsekas, D. and Gallager, R. *Data Networks*, Second Edition. Prentice Hall, 1992.

[4] Cheng, S., Garlan, D., Schmerl, B., Sousa, J., Spitznagel, B., Steenkiste, P., and Hu. N. Software Architecture-based Adaptation for Pervasive Systems. Proc. International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing, April 8-11, 2002. Lecture Notes in Computer Science, Vol. 2299, Schmeck, H., Ungerer, T., Wolf, L. (Eds).

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

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

[7] Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. Proc. SIGSOFT '94 Symposium on the Foundations of Software Engineering, New Orleans, LA, Dec. 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.R., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. Proc. Working Conference on Complex and Dynamic System Architecture. Brisbane, Australia, 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] Gorlick, M.M. Distributed Debugging on $5 a day. Proc. the California Software Symposium, University of California, Irvine, CA, 1997 pp. 31-39.

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

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

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

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

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

[17] 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, 1997.

[18] 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.

[19] 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.

[20] Oriezy, P., Medvidovic, N., and Taylor, R.N. Architecture-Based Runtime Software Evolution. Proc. International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, Apr. 1998.

[21] 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.

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

[23] 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.

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

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

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