Model-based Adaptation for Self-Healing Systems

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
412 268-5056
garlan@cs.cmu.edu

Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
412 268- 5889
schmerl@cs.cmu.edu

ABSTRACT
Traditional mechanisms that allow a system to detect and recover from errors are typically wired into applications at the level of code where they are hard to change, reuse, or analyze. An alternative approach is to use *externalized adaptation*: one or more models of a system are maintained at run time and external to the application as a basis for identifying problems and resolving them. In this paper we provide an overview of recent research in which we use architectural models as the basis for such problem diagnosis and repair. These models can be specialized to the particular style of the system, the quality of interest, and the dimensions of run time adaptation that are permitted by the running system.

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

General Terms
Design, Measurement.

Keywords
Self-healing systems, software architecture, model-based systems.

1. INTRODUCTION
Research in software engineering has traditionally been based on the implicit assumption that systems are created and modified off-line. However, increasingly systems are required to work continuously, and to do so in environments where users' requirements and system resources may change frequently. To address this new kind of capability software engineers will need cost-effective techniques and mechanisms to build systems that reliably and adapt their own behavior dynamically.

Mechanisms that that support self-adaptation have been around for a long time in the form of programming language features (e.g., exceptions and run-time assertion checking) and algorithms (e.g., network protocols, self-stabilizing algorithms). Most adaptation mechanisms found in existing systems, however, are tightly integrated with the application itself and wired in at the code level. Such "internal" 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 or assertion checking) and run-time libraries (e.g., timeouts for RPC). However, they suffer from the problem that localized error handling may not be able to determine the true source of the problem, and hence the required remedial action. Moreover, while they can trap errors, they are not well-suited to recognizing "softer" system anomalies, such as gradual degradation of performance, or patterns of unreliability. Finally, they make it difficult to change adaptation policies, because they are so intertwined with the normal code of the system.

An alternative approach is to "externalize" adaptation. Externalized adaptation supports a kind of closed-loop control system paradigm illustrated in Figure 1. In this paradigm system behavior is monitored by components outside the running system. These components are responsible for (a) determining when a system's behavior is within the envelope of acceptable system parameters, and (b) when it falls outside of those limits, adapting the system. To accomplish these tasks, the externalized mechanisms maintain one or more system models[1], which provide an abstract, global view of the running system, and support reasoning about system problems and repairs.

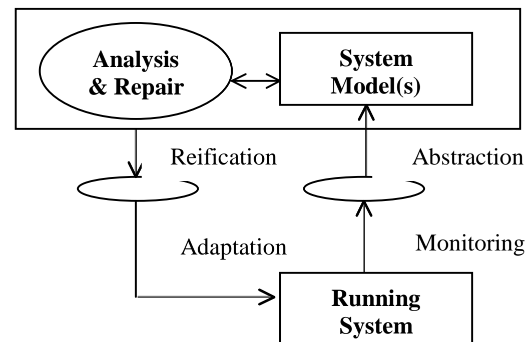

Figure 1. Model-based Adaptation

This approach is attractive for a number of reasons: First, different models can be chosen, depending on the system quality of interest. Second, externalized mechanisms can support reuse, since they are not application-specific. Third, they can be easily changed, since they are localized. Fourth, they can exploit a large

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-000-0/00/0000 ...$5.00

[1] By "model," we mean an abstract representation of a system. Examples include architecture models, performance models, reliability models, etc.

body of existing work on analytical methods for improving attributes such as performance, reliability, or security.

However, from a software and systems engineering perspective externalized, model-based adaptation raises a number of challenging research problems.

Monitoring: How do we add monitoring capabilities to systems in non-intrusive ways? What kinds of things can we monitor? Is it possible to build reusable monitoring mechanisms that can be easily added to existing systems? How should we design components and systems so that they can be more easily monitored?

Interpretation: How do we make sense of low-level monitored information? How can we determine when there is a problem that requires system adaptation or, more generally, an opportunity for improving a system through adaptation? How can we pinpoint the source of a problem? What models are best paired with specific quality attributes and systems?

Resolution: How do we repair a system, once we know there is a problem? How do we select the best repair action from a set of possible repairs? Can we guarantee that our repairs will actually improve things? How do we reconcile conflicting repairs obtained multiple models? Can we "improve" a system even when no specific error has arisen?

Adaptation: How do we cause the adaptation to occur in a running system? How should we design/build software systems and components so that they can be dynamically adapted? What do we do if something goes wrong during the process of adaptation?

In addition, there are a number of general open issues associated with the approach. Given the overheads associated with monitoring, interpretation, resolution and adaptation, what kinds of problems and behaviors are best suited to such an externalized approach? To what extent can we minimize the costs of making systems self-adaptive? Does the approach scale to Internet-based systems and services, where we may have limited control over an application's constituent components?

In this paper we sketch answers to a subset of these questions, based on our perspective of research over the past three years on the Rainbow Project at Carnegie Mellon. Specifically, we describe how we have instantiated the general paradigm outlined above. The key idea is to use style-specific architectural models as the basis for interpretation, together with a reusable, easily-tailored infrastructure for monitoring and resolution.

2. RELATED WORK

The idea of model-based adaptation has existed for some years in a variety of contexts. Most of these have focused on the use of specific models (e.g., performance models to support load balancing), rather than the more general issue of software engineering support for externalized adaptation.

The use of architectural models as the centerpiece of model-based adaptation has been explored by a number of researchers [18]. These systems have typically focused on the use of specific styles to provide intrinsically-modifiable architectures. Taylor and colleagues use hierarchical publish-subscribe via C2 [17]; Gorlick and colleagues use data-flow style via Weaves [12]; and Magee and colleagues use bi-directional communication links via Darwin [14]. Wermelinger and colleagues [24] use architectural primitives, independent of particular architectural styles, to effect architectural changes.

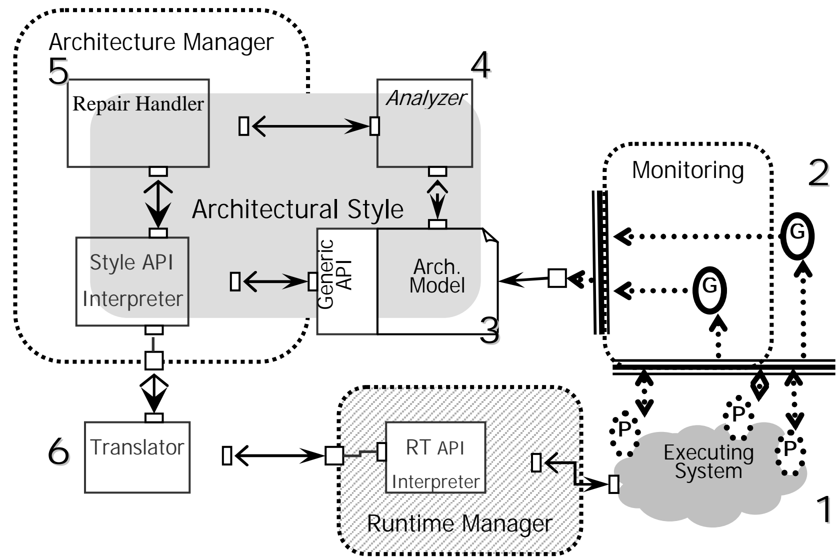

Figure 2. Adaptation Framework.

As we describe below, our own work has focused on the use of architecture styles as an explicit design element. In previous publications we have described various aspects of this work: the use of styles [7], reusable monitoring infrastructure [4] and other tools [20], applications to pervasive computing [5], and experimental evaluation [6]. This paper provides a high-level overview of this previously published research.

3. OVERVIEW OF THE APPROACH

Our approach to self-adaptation, is illustrated in Figure 2. An executing system (1) is monitored to observe its run time behaviour (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 centerpiece of the approach is the use of architectural models [19][21]. We use a simple scheme in which an architectural model is represented as a graph of interacting components. This is the core architectural representation scheme adopted by a number of architecture description languages (ADLs), including Acme [10], xADL [8], and SADL [16]. Nodes in the graph are termed *components*. They represent the principal computational elements and data stores of the system: clients, servers, databases, user interfaces, etc. Arcs are termed *connectors*, and represent the pathways of interaction between the components. A given connector may in general be realized in a running system by a complex base of middleware and distributed systems support.

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

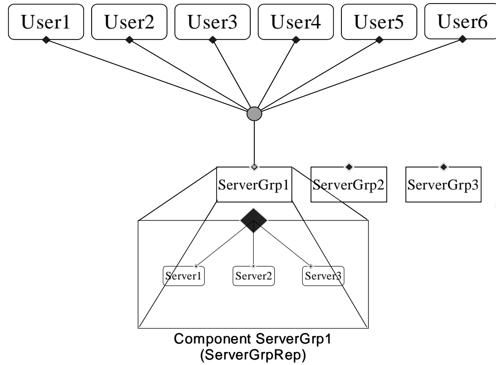

Figure 3. Example Client Server System

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][21][22][23].

To illustrate how adaptation exploits architectural models, consider a 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 3). 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 architecture is illustrated in Figure 3.

Let us assume that the organization managing the web service infrastructure wants to make sure that two 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.

We will also assume that the system has two built-in low-level adaptation mechanisms. First, it is possible to activate a new server in a server group or deactivate an existing server. Second, we can cause a client's communication path to change 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, we accomplish this as follows: First, given the nature of the implementation, we 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. Next, because we are focusing on performance, we adapt that style so that it exposes performance-related properties and makes explicit constraints about performance. 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. (See [7] for details.)

```
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 4. Simple Client-Server Style

Figure 4 outlines the definition of the base client-server style, written in Acme, and Figure 5 shows how this style can be extended to include performance-oriented constraints.

To make the style useful as a run time artifact we 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). Figure 6 illustrates a repair script associated with the constraint on average latency for client requests.

```
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 5. Client-Server Style Extended for Performance Analysis

There are now only two remaining problems. First, we must get information out of the running system. To do this we use 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 [4]. For example, gauges aggregate 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 use a table-driven translator that interprets 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. (See for details.) Moreover, matching the architectural style to the existing system infrastructure helps guarantee that relevant information can be extracted, and that architectural changes can be propagated to the running system.

4. IMPLEMENTATION

Our current implementation is based on the Acme language and its accompanying AcmeStudio toolset. AcmeStudio was originally developed as a design-time architect's assistant. To make it function as a run time tool, we provided a COM interface to accept events received from the monitoring infrastructure.

Changes to architectural properties are received via the monitoring infrastructure, which is implemented in Java, and uses the Siena wide area event bus to communicate messages between components. Gauges are implemented using a gauge library which uses a gauge protocol that we have defined for gauge creation, communication, and deletion.

Probes in the implementation and environment use the Remos Monitoring System [13] and a set of application-specific probes. The application-specific probes are implemented using AIDE [2], which preprocesses Java source code to facilitate the instrumentation of the code. The probes report when particular methods have been called, so that, for example, bandwidth, latency, and server load can be calculated by the gauges. These events are also reported to a Siena bus. Currently, we use hand-tailored support for translating APIs in the *Model Layer* to the *Runtime Layer*.

Architectural constraints are checked in the running system via a tool, called Armani, which evaluates first order constraints (much in the style of UML's OCL) on the fly as properties of the architecture change. When problems are detected Armani triggers a repair engine, called Tailor, to look for a repair strategy.

```
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. An Example Repair Strategy.

5. EXPERIENCE

Thus far we have experimented with architectural adaptation for two kinds of system properties: (1) performance for web-based systems, illustrated earlier, and (2) protocol conformance.

To evaluate the effectiveness of our adaptation framework for performance-oriented adaptation, we conducted an experiment to test system adaptation using a dedicated, experimental testbed consisting of five routers and eleven machines communicating

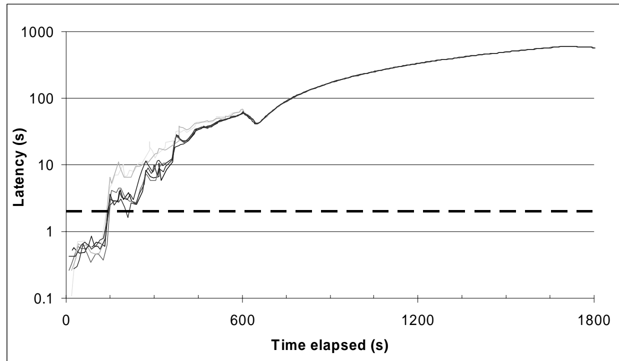

Figure 7. Average Latency for Control.

over 10 Mbps lines. The implementation that we used for our experiment was based on the example presented in this paper – that of a client-server system using replicated server groups communicating over a distributed system. System loads were constructed by playing back monitored system behavior collected during actual use at CMU on the main campus network, modified to introduce regular increases in usage so we could observe the self-repair behavior of the system.

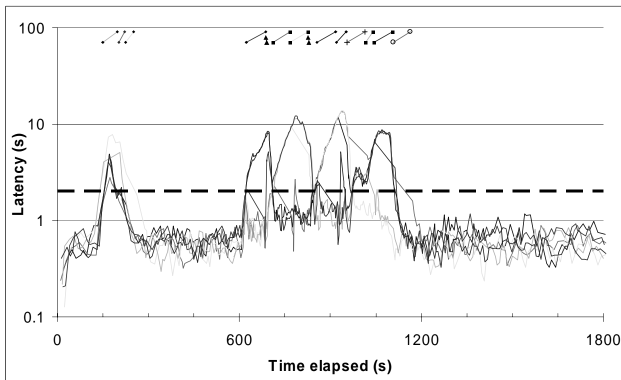

Figure 8. Average Latency under Repair.

The results showed that for this application and the specific loads used in the experiment, self repair significantly improved system performance. Figures 7 and 8 show sample results for the system performance without adaptation, and with, respectively. (See [6] for details.) However, it also revealed, perhaps not unexpectedly, that externalized repair introduces some significant latency. In our system it took several seconds for the system to notice a performance problem and several more seconds to fix it. Although we can imagine speeding up the roundtrip repair time, this does indicate that the approach is best suited for repair that operates on a global scale, and that handles longer term trends in system behavior.

The second application of the approach has been to monitor and check protocols of interaction between components. Connectors are associated with protocol constraints that indicate the allowed order of communication events. These are defined in a process algebra, FSP [15], and then used by "protocol gauges" at run time to detect when communicating components fail to respect the specified protocols. For example, a protocol error might occur when a component attempts to write data to a pipe after it has closed that pipe, or if a client attempts to communicate with a server without first initializing its session.

6. CONCLUSIONS AND FUTURE WORK

Externalized self-repair based on the use of models appears to be a promising approach. In this paper we outlined our use of that paradigm using architectural models as the basis for monitoring, problem detection, and repair. That is, architectural models are retained at run time as a way to understand what the running system is doing in high level terms, detect when architectural constraints are violated, and reason about repair actions at the architectural level.

One of the main premises of our work is that considerable benefit can be obtained by using models that are style-specific. The use of style provides a focused context through which to understand a system and fix it.

However, one style does not fit all. For different properties and different kinds of systems, different styles will be relevant. Our main line of research has focused on distributed client-server architectures with strong performance requirements. But other applications, such as automotive control, or information management systems will most naturally be represented using quite different style, and associated repair policies.

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 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 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).

7. ACKNOWLEDGMENTS

This work is supported in part by DARPA under Grants N66001-99-2-8918 and F30602-00-2-0616 and by the High Dependability Computing Program from NASA Ames, cooperative agreement NCC-2-1298.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 or NASA.

8. REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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).

[6] Cheng, S., Garlan, D., Schmerl, B., Steenkiste, P., and Hu. N. Software Architecture-based Adaptation for Grid Computing. The 11th IEEE Conference on High Performance Distributed Computing (HPDC'02), Edinburgh, Scotland, July 2002.

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

[8] 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.

[9] 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.

[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. Proc. Working Conference on Complex and Dynamic System Architecture. Brisbane, Australia, December 2001.

[12] 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.

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

[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] Magee, J. and Kramer, J. *Concurrency: State Models and Java Programs*. Wiley 1999.

[16] 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.

[17] 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.

[18] 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.

[19] Perry, D.E., and Wolf, A. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4):40-52, October 1992.

[20] Schmerl, B. and Garlan, D. Exploiting architectural design knowledge to support self-repairing systems. The 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 15-19, 2002.

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

[22] 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.

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

[24] 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.