Proceedings of the European Workshop on Software Architectures,
St. Andrews, Scotland, May 2004.

Using Architectural Models at Runtime: Research Challenges

David Garlan and Bradley Schmerl

Department of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213 USA
{garlan,schmerl}@cs.cmu.edu

1. Introduction

One crucial aspect of high quality software engineering is the development of a well-defined software architectural model. Such a model describes the runtime manifestation of a software system in terms of its high level components and their interconnections. A good architectural model can be used as the basis for design-time analysis to determine whether a software system will meet desired quality attributes.

Despite advances in using software architectural model to clarify system design, there remains a problem that is typical of design-time artifacts: Does the system as implemented have the architecture as designed? Without some form of consistency guarantee the relationship between the architecture and the implementation will be hypothetical, and many of the benefits of using an architecture in the first place will be lost. One approach to addressing this problem is to enforce correspondence by generating code from the architectural model or by forcing developers to implement against a specific code library, which can then be used to provide some guarantees (e.g., [1,8,10]). Another approach is to use static code analysis techniques to determine the architecture of the code, subject to some constraints about code modularization and code patterns [5,6,7].

An alternative approach is to monitor the running system and translate observed events to events that construct and update an architectural model that reflects the actual running system. One can then compare this dynamically-determined model to the correct architectural model. Discrepancies can be used to flag implementation errors, or, possibly, to effect run-time adaptations to correct certain kinds of flaws.

At Carnegie Mellon University, our research group has been investigating the use of system monitoring and reflection using architectural models. In the process of exploring this area we have identified a number of significant research challenges. In this paper we outline our experience, and use that as a way to lay out an agenda for architecture-based approaches to system monitoring and system self-repair. We then briefly outline the ways in which we have been addressing some of these challenges.

2. Research challenges

The notion of using architecture models at runtime to monitor and repair a running system is attractive for a number of reasons: First, different architectural models or views can be chosen depending on the system quality of interest. Second, externalized mechanisms can support reuse, since they are not application-specific. Third, the details of how models are derived and of what to do if something is wrong can be easily modified, since they are localized in the external mechanisms and not distributed throughout the application. Fourth, the models used as the basis for external reasoning can exploit a large body of existing work on analytical methods for improving attributes such as performance, reliability, or security.

However, achieving these benefits requires that one address a number of research challenges. These challenges can be divided into four categories:

1. **Monitoring.** How do we add monitoring capabilities to systems in non-intrusive ways? What kinds of things can be monitored? Is it possible to build reusable monitoring mechanisms that can be added to existing systems?
2. **Interpretation.** How do we make sense of monitored information? How do we produce architectural models from this information? How can we determine whether a problem exists with the running system and whether repair, or more generally improvement, is required? How can we pinpoint the source of a problem? What models are best paired with specific quality attributes and systems?
3. **Resolution.** Once we know there is a problem – that the running system is at variance with the intended architectural design and its behavior – how do we decide what to do to fix it? How can we select the best repair action from a set of possible actions? What can we guarantee about a repair? Can we "improve" a system even if there is no specific problem?
4. **Adaptation.** How can we cause the adaptation to occur in a running system? What do we do if something goes wrong during the process of adaptation? How do we know that the adaptation actually worked to repair the system?

Ideally solutions to these problems would lead to mechanisms that not only add new capability to existing systems, but do so in a cost-effective manner. That is, we would like to find reusable infrastructure that addresses many of these issues, and have ways to adapt that infrastructure to specific systems.

3. Experience with architecture-based monitoring and repair

In an attempt to gain some experience with these issues we have been exploring the use of architectural models at run time in the context of a project called Rainbow [2]. To address issues of cost-effectiveness, our approach to providing dynamic architecture discovery and repair is to provide an "externalized" *generic infrastructure* that is

independent from an executing system and that can be *specialized* for particular target systems. Such an approach allows us to target existing systems for which (a) the code was not written with any particular convenient library or code pattern; (b) an architectural model may not exist; or (c) adaptation was not designed a priori.

The externalized approach supports a form of closed-loop control system, where system behavior is monitored, analyzed, and (if required) adapted. In such a case, the architectural model acts as a basis for reasoning about the observations of the system, and also for reasoning about changes that may need to be made.

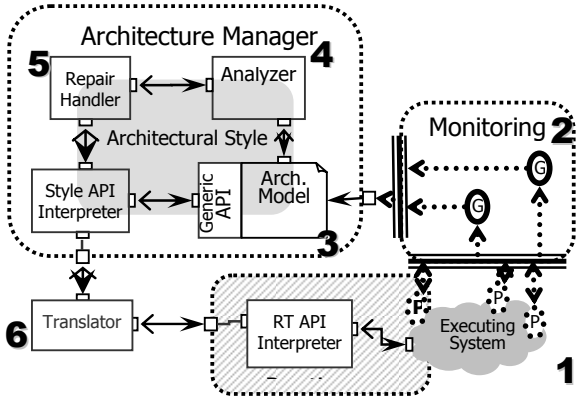

Figure 1. Adaptation Framework.

The approach is illustrated in Figure 1. In the Rainbow framework, architectural models and styles are central to providing externalized self-adaptation mechanisms. An executing system is (1) is monitored to observe its run time behavior (2). Monitoring information is abstracted and related to architectural properties and elements in an architectural model (3). Changes in the architectural model trigger rule evaluation (4) to determine whether the system is operating within an envelope of acceptable ranges. Violations of rules are handled by a repair mechanism (5), which adapts the architecture. Architectural changes are propagated to the running system (6).

Our approach to dynamically discovering an architecture and issuing repairs based on observations about that architecture requires a generic framework that can be used in many systems, and a means of specializing this framework for particular domains to effect useful and meaningful discovery and repair in that domain.

The specialization of the framework requires us to specify many parts. A key challenge is how to maximize reuse so that details can be shared in the same domain. For example, if we are detecting and adapting systems in the domain of automotive software, we should be able to reuse many of the details regardless of the system. Part of this challenge is identifying what can be reused, and when, in a methodical manner.

3.1 Architectural Style

Key to solving the reuse challenge is the use of architectural style to parameterize our generic infrastructure. We consider an architectural style to be a collection of types for components, connectors, interfaces, and properties together with a set of rules for how elements of those types may be composed. Properties are associated with elements in the architecture to capture behavior and semantics. For example, a property on a connector type might be used to indicate its protocol or capacity. Rules can, for example, specify that some property value must be within certain ranges.

The benefits of associating a style with an architecture include support for analysis, reuse, code generation, and system evolution [3,9,10]. In particular, the knowledge, rules, and analyses can be defined at the style level and reused in all instances of that style. This has proved extremely useful at design-time for providing tools to guide an architect in developing a model. We are attempting to exploit this reuse for dynamic repair by factoring repair and monitoring facilities that will be common for all architectures of a particular style and specializing our generic infrastructure with these facilities to provide repair infrastructure for systems of a particular architectural style.

To make the style useful as a runtime artifact for repair requires us to extend the traditional notion of architectural style with two more concepts:

1. A set of style-specific *architectural operators* that can be used to change an architecture in that style. Such operators are more than just simple operations for adding or removing architectural elements; they are written in terms of the vocabulary for the style and should result in models that conform to the architectural style. For example, an operation to add a client in a client-server style would also involve connecting the model to a server. Removing a server may relocate or delete clients.

2. A collection of *repair strategies* written in terms of these operators associated with the style's rules. If a dynamic observation is interpreted as violating a rule of the architecture, then a repair is issued which uses properties of the style to pinpoint the error, and operators of the style to adapt the architecture.

The operators and repair strategies are chosen based on an examination of the analyses associated with a style, which formally identify how the architecture should change in order to affect desired characteristics.

The key to making this work is to parameterize the Architecture Manager (Figure 1) with an architectural style. Within a style, or domain, the Architecture Manager will remain largely unchanged – the analyzer will analyze the same rules, the Style API will use the same operators, and the repair handler will likely use the same repair strategies or tactics. To reuse the infrastructure in another domain requires specializing the framework with a different architectural style.

A second critical issue is then getting the information out of the system and effecting changes back into the system. To address the *Monitoring* challenges, we divide the problem into system-level information, which can be ascertained by using off-the-shelf probing technologies (such as network monitors, debugging interfaces, etc.), and architecture level information. To bridge the gap between system-level information and architectural properties we use a system of adapters, called "gauges," which ag-

gregate low-level monitored information and relate it to the architectural model [4]. For example, gauges may aggregate measurements of the round-trip time for a request and the amount of information transferred to produce bandwidth measurements at the architectural level. Gauges thus interpret monitored events as properties of an architectural model.

To address the challenge of *Adaptation*, we use a knowledge base to map architecture operations into system-level operations to make changes to the system. This knowledge base uses customized translations in addition to collecting information from gauges.

3.2 Architecture discovery

Until recently, gauges in our work were restricted to monitoring properties of architectural models. They were used merely to monitor the system and interpret those observations as properties on a pre-existing architecture. In order to address the challenge of determining the architecture of a running system, and to help determine whether architectural repairs have been enacted in the system, we need a method for taking observations about the running system and discover its architectural structure.

DiscoTect [11] is a system for discovering the architectures of running, object-oriented systems and can be used to construct architectures dynamically. The novelty of DiscoTect is the way that the mapping between the system and the architecture specified. A form of state machine is used to keep track of the progress of system-level events and emit architectural events when combinations of system-level events are detected. We require a state machine because a given architectural event, such as creating a connector, might involve many runtime events. Conversely, a single runtime event might correspond to multiple architectural events. For example, a simple method invocation may signal the creation of a connector, its associated interfaces, and connecting the connector to particular components. Complicating this further is the fact that many architectural events may be interleaved in an implementation. For example, a system might be midway through creating several components and connectors.

Again, the notion of style is helpful in providing reuse for this complicated process. The architectural events will be in terms of the operators of the style. We may also be able to take advantage of particular pairings of architectural style and implementation conventions to garner common parts of the state machine, thus generalizing detection more. For example, if the implementation is written in CORBA, many CORBA events will map to the same architectural events for a particular architectural style.

4. Conclusion

In this position paper, we outlined a set of challenges that need to be addressed in order to make architectures available and useful at runtime. We argued that using architectural information dynamically has benefits of providing a feasible and flexible approach for discovering a system's architecture, and for detecting faults, reasoning

about them, and deciding repairs. We then indicated some of the challenges that fall within the categories of monitoring, interpretation, resolution, and adaptation. Next we outlined research that we believe addresses some of those challenges. As we have tried to indicate, architecture-based monitoring and adaptation is a rich area of ongoing research, and ripe for contributions along many lines, from engineering to foundations.

References

1. Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting Software Architectures to Implementation. In Proc. 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL., pp. 187-197, 2002.
2. Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J., Spitznagel, B., Steenkiste, P. Using Architectural Style as the Basis for Self-repair. Proc. the 3rd Working IEEE/IFIP Conference on Software Architecture, pp. 45-59, 2002.
3. Garlan, D., Allen, R., Ockerbloom, J. Exploiting Style in Architectural Design. Proc. SIGSOFT'94 Symposium on the Foundations of Software Engineering, New Orleans, 1994.
4. Garlan, D., Schmerl, B., and Chang, J. Using Gauges for Architecture-based Monitoring and Adaptation. Proc. Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 2001.
5. Jackson, D., WainGold, A. Lightweight extraction of object models from byte-code. In Proc. 21st International Conference on Software Engineering, Los Angeles, CA, 1999.
6. Kazman, R., Carriere, S. Playing Detective: Reconstructing Software Archtiecture from Available Evidence. Journal of Automated Software Engineering 6(2):107-138, 1999.
7. Murphy, G., Notkin, D., Sullivan, K. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In Proc 13th ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington D.C., pp. 18-28, 1995.
8. Shaw, M., Deline, R., Klein, D., Ross, T., Young, D., Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering (TOSEM) 21(4):314-335, 1995.
9. Shaw, M. and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
10. Taylor, R., Medvidovic, N., Anderson, K., Whitehead, E., Robbins, J, Nies, K., Oriezy, P., Dubrow D. A Component- and Message-based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6):390-406, 1996.
11. Yan, H., Garlan, D., and Schmerl, B. DiscoTect: A System for Discovering Architectures from Running Systems. In Proc. 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004.