

The Architect

The Recovery of Runtime Architectures

Rick Kazman, Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich

An increasingly important requirement for software-based systems is their ability to adapt at run time to handle such things as resource variability, changing user needs, changing demands, and system faults. In the past, systems that supported such self repair were rare, confined mostly to domains such as 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 ecommerce systems and mobile embedded systems. For systems to adapt themselves, one of the essential ingredients is *self reflection*: a system must know what its status is, and it must be able to identify opportunities for improving its own behavior.

Traditionally software systems have operated in relatively stable, fixed environments (such as a desktop) and could be taken down for maintenance, upgrading, or replacement. However, software systems must increasingly function in environments where resources (such as wireless bandwidth) change rapidly, where their resource demands are difficult to predict, where they must interact with potentially faulty components and services not under their control, and yet where they must continue to operate continuously. In short, such systems must begin to take more responsibility for their own health and welfare, adapting at run time to handle errors, changing resources, and varying user needs.

Today software engineers have few tools or techniques to create such self-adaptive systems reliably, flexibly, and at low cost. Most techniques at our disposal rely on low-level mechanisms such as exceptions and timeouts. But these mechanisms generally provide little help in allowing a system to determine the true source of problems or to choose a response to them. Moreover, they are ineffective at dealing with softer problems, such as gradual performance degradation, or at recognizing opportunities to improve behavior even when things are not broken.

For the past year, we have been investigating a new paradigm for software systems that is already showing promise for solving this problem. The underlying idea is to associate with each software system a *reflective architecture model* that allows a system to reason about its own behavior at run time and take action to modify its own structure and behavior when necessary. By reflecting the current state of a system as an architectural model that exposes only the main components, interactions, and their high-level properties, a system can more easily understand what its current state is and take necessary actions.

A critical step toward achieving this vision is the ability to know exactly what the architecture of a running system is. We use run-time monitoring and abstraction, together with codified knowledge about architectural styles, to develop a dynamic view of a system's architecture as it runs. In this way, we can instrument a system with *probes* that produce streams of low-level system observations that are then in-

terpreted by a rule-based abstraction engine to produce higher-level architectural events and operations reflecting the system's architecture as it is running. (See Figure 1.)

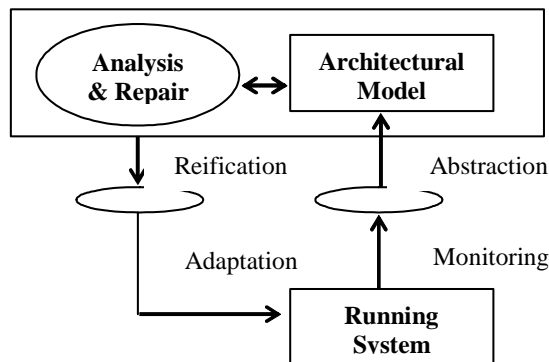


Figure 1 Architecture-Based Adaptation

Currently two techniques have been used to determine or enforce relationships between a system's architecture and implementation. The first is to ensure consistency by construction. This can be done by embedding architectural constructs in an implementation language where program analysis tools can check for conformance. Or it can be done through code generation, using tools to create an implementation from a more abstract architectural definition. While effective when it can be applied, this technique has limited applicability. In particular, it can usually be applied only in situations where engineers are required to use specific architecture-based development tools, languages, and implementation strategies. For systems that are composed out of existing parts or that require a style of architecture or implementation outside those supported by generation tools, this approach does not apply.

The second technique is to ensure conformance by extracting an architecture from a system's code using static code analysis (for more information see http://www.sei.cmu.edu/ata/ata_extraction.html). When an implementation is sufficiently constrained that modularization and coding patterns can be identified with architectural elements, this can work well. Unfortunately, however, the technique is limited by an inherent mismatch between static, code-based structures (such as classes and packages) and the run-time structures that are the essence of most architectural descriptions. In particular, the actual run-time structures may not even be known until the program runs: clients and servers may come and go dynamically; components not under direct control of the implementers may be dynamically loaded; etc.

A third technique—the one we are following here—is to determine the architecture of a system by examining its behavior *at run time*. Observations about its behavior can then be used to infer its dynamic architecture. This approach has the advantages that in principal it applies to *any* system that can be monitored, it gives an accurate image of what is actually going on in the real system, it can accommodate systems whose architecture changes dynamically, and it imposes no a priori restrictions on system implementation or architectural style.

There are a number of hard technical challenges in making this technique work. The most serious is finding mechanisms to bridge the abstraction gap: in general, low-level system observations do not map directly to architectural actions. For example, the creation of an architectural connector might involve many low level steps, and those actions might be interleaved with many other architecturally relevant actions. Moreover, there is likely no single architectural interpretation that will apply to all systems: different systems will use different runtime patterns to achieve the same architectural effect, and conversely, there are many possible architectural elements to which one might map the same low level events.

We have developed a technique to solve the problem of dynamic architectural discovery for a large class of systems. The key is to provide a framework that allows the mapping of implementation styles to architecture styles. This mapping is defined as a set of conceptually concurrent state machines that are used at run time to track the progress of the system and output architectural events when predefined run time patterns are recognized. By parameterizing the framework by both architectural and implementation styles, we are able to exploit regularity in systems while still providing flexibility in defining new abstraction mappings.

The system we have built to do this is called DiscoTect [Yan 04]. Any approach that supports dynamic discovery of architectures must be able to (a) observe a system's runtime behavior, (b) interpret that runtime behavior in terms of architecturally meaningful events, and (c) represent the resulting architecture. In DiscoTect we are primarily concerned with the second problem of bridging the abstraction gap between system observations and architectural effects.

In DiscoTect we adopt an approach illustrated in Figure 2. Monitored events are first filtered by a trace engine to select the subset of system observations that must be considered. The resulting stream of events is then fed to a state engine. The heart of this recognition engine is a state machine designed to recognize interleaved patterns of runtime events and, when appropriate, to output a set of architectural operations. Those operations are then fed to an architecture builder that incrementally creates the architecture, which can then be displayed to a user or processed by architecture-analysis tools.

To handle the variability of implementation strategies and possible architectural styles, we provide a language to define new mappings. Given a set of implementation conventions (an *implementation style*) and

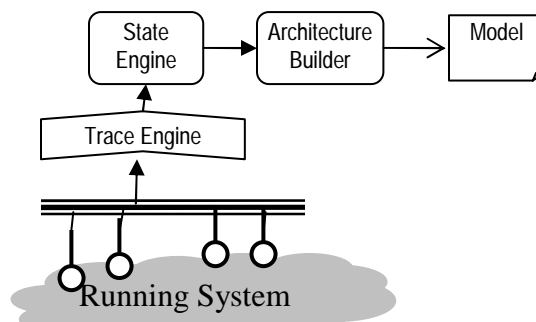


Figure 2. The DiscoTect Architecture

a vocabulary of architectural element types and operations (an *architectural style*), we provide a description that captures the way in which runtime events should be interpreted as operations on elements of the architectural style. Thus each pairing of implementation style and architectural style has its own mapping. A significant consequence is that these mappings can be reused across programs that are implemented in the same style.

We have used DiscoTect to recover the architecture of a number of systems, the largest of which are the JBoss J2EE Server (<http://www.jboss.org>) and Sun's J2EE Server (<http://java.sun.com/j2ee/>). Such is the generality of our approach that we were able to reuse most of the effort in creating the JBoss mapping when we analyzed Sun's J2EE Server.

We are now working on recovering the architecture of systems written in C and C++, to test the generality of our approach. Early results from this work appear promising. DiscoTect is showing tremendous potential as a tool to recover and monitor run-time architectures, a critical step in realizing architecture-based run-time adaptation.

Reference

- [Yan 04] Yan, H.; Garlan, D.; Schmerl, B.; Aldrich, J.; & Kazman, R. "DiscoTect: A System for Discovering Architectures from Running Systems," *Proceedings of the 26th International Conference on Software Engineering (ICSE 26)*, Edinburgh, Scotland, May 23-28, 2004.

About the Authors

Rick Kazman is a senior member of the technical staff at the SEI, where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He is also an adjunct professor at the Universities of Waterloo and Toronto. His primary research interests within software engineering are software architecture, design tools, and software visualization. He is the author of more than 50 papers and co-author of several books, including a book recently published by Addison-Wesley titled *Software Architecture in Practice*.

Kazman received a BA and MMath from the University of Waterloo, an MA from York University, and a PhD from Carnegie Mellon University.

David Garlan is an Associate Professor in the School of Computer Science at Carnegie Mellon University, where he leads several research projects and is the Director of the Master in Software Engineering Program. He received his Ph.D. from Carnegie Mellon University in 1987. Dr. Garlan's research interests include software architecture, ubiquitous computing, self-adaptive systems, formal methods, and software development environments. Dr. Garlan and his research group have developed a number of lan-

guages and tools for design of software architectures including: Wright (a formal language for software architectures that focuses on specification and analysis of component interactions) and Acme (a language and design environment for software architecture, supporting rapid customization to architectural styles). His recent research has focused on developing engineering principles and techniques for building self-healing and self-adaptive systems, a fundamental capability of modern software systems. Dr. Garlan has written dozens of papers on software architecture, and co-authored two books: "Software Architecture: Perspectives on an Emerging Discipline," and "Documenting Software Architectures: Views and Beyond."

Bradley Schmerl is a Systems Scientist in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. from Flinders University of South Australia in 1997. Dr. Schmerl's research interests include software architectures, self-adaptive systems, software development environments, and ubiquitous computing. Together with Prof. Garlan, he has developed tools to aid in the design of software architectures.

Jonathan Aldrich is an assistant professor at the Institute for Software Research International at Carnegie Mellon University, working in the areas of software engineering and programming languages. At CMU he leads the ArchJava project, which integrates a software architecture specification into the Java language, using type-based techniques to ensure that the code respects architectural constraints. Aldrich received a B.S. from the California Institute of Technology and a Ph.D. from the University of Washington.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.