Higher-Order Connectors

Presented at the
Workshop on Compositional Software Architectures
Monterey, CA, January 6-7, 1998

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A critical issue for architectural design is the nature of the glue, or connectors, with which a system's parts are combined. Thus an important first step toward improving our ability to compose parts is to make to make connectors explicit semantic enties, where they can be documented, analyzed, and (sometimes) used to generate code. A number of notations for software architecture do precisely this. However, a key second step is to understand *operations over connectors.* In principle, such operations would permit one to produce new connectors out of old ones, adapt existing connectors to new contexts of use, and factor out common properties of connectors so they can be reused. In this paper we argue that the use of "higher order connectors" is one way to achieve this goal.

1 Motivation

To compose a system from parts a key design consideration is to make sure the components can interact using the communication mechanisms at hand. Communication mechanisms – or *connectors* – can be as simple as procedure call and data sharing (typically provided by a programming language or operating system), or as complex as secure protocols for distributed transactions (often provided by middleware products).

In the past the problem of achieving interoperability between a set of components has been cast as a problem of adapting the components to match some fixed set of connector types. For example, in Unix components are typically programmed to read and write from pipes. With RPC or Corba, components must be written to communicate via procedure calls via stubs and proxies. With event broadcast components must be capable of receiving and responding to asynchronous messages. In this scenario, when a component does not match a given connector, one is forced to rewrite the component, or possibly add a wrapper to adapt the component for its context of use.

However, there is a different way to attack the problem: rather than adapting the components, adapt the connectors. That is, find ways to massage a given set of connectors into a form that permits the existing components to interoperate. While this approach may not always be possible, it holds promise as a powerful technique for system composition: since many components can't be rewritten, and wrappers may not be sufficiently powerful, adapting the connectors may be the best option at hand.

Consider the following simple example. Suppose you would like to add capabilities for monitoring error messages that are communicated between components in some system. One option is to rewrite each component to send a copy of each error message to a monitoring component whenever those errors arise. A better plan might be to provide a set of component wrappers that observe whether outgoing messages are error messages, and forward a copy to the monitor component.

However, another way is to consider adapting the connectors so that they automatically monitor error messages, and send them to the appropriate entity in the system for processing. Since there are typically many fewer connector types in a system than component types, this has significant cost savings. But perhaps more importantly it leads to a new opportunity for understanding not only ways to modify connectors, but *principled* ways of doing so.

In this paper we sketch one avenue for making the notion of connector adaptation principled. Specifically, we argue that it is possible and desirable to consider classes of connector operators: that is, operators that take connectors as parameters and produce connectors as results. By analogy to functions that allow functions as their arguments and results, we call these *higher-order connectors*. Further, we will argue that there are good reasons to believe that higher-order connectors can support sound semantics, useful algebraic theories, and capabilities for implementation guidance and generation.

2 Connectors as first class design entities

An important first step in understanding how to integrate components is to permit the explicit description and analysis of connector types [Sha93]. By making connector abstractions explicit, one can make principled design choices between different interaction schemes, support analysis of those interactions, and, in some cases, automatically generate implementations for those interactions.

A number of architecture description languages have therefore taken this approach, including Wright [AG94], UniCon [SDK+95], Acme [GMW97], and SADL [MQ94]. These languages permit the designer to choose among a rich selection of connector types, and (in many cases) to specify new kinds of connector types.

Unfortunately, in all of these languages, connectors are either provided as primitives (albeit, often a flexible set), or must be defined from scratch. For example, in UniCon, while one can add new connectors to the system, the task of doing so requires detailed knowledge of the architectural tools and representations. In practice, this makes it difficult to introduce new connector types, and limits our ability to understand relationships between different connector types. For example, one might like to in what contexts a shared memory buffer connector is equivalent to a pipe.

3 Higher-order connectors

A solution to this problem is to provide operators with which new connectors can be built up from old connectors using well-founded operators for composition.

Consider the following examples.

1. *Bundling*: A set of procedure calls is bundled into a single entity, sometimes called an API.
2. *Monitoring*: A connector is modified to transmit certain classes of communication events to a monitoring component.
3. *Confirmation*: An RPC connector is modified to permit the callee to ask for confirmation from some third party when an dubious operation is attempted.
4. *Security*: An arbitrary connector is augmented with encryption, or possibly authentication, facilities.
5. *Compression*: A connector is modified to transparently compress its data for transmission.

4 Challenges

Each of the examples above has the form of producing a new connector type from one or more existing connector types (together with certain other information). By focusing on the problem in this way we can begin to ask some important questions:

- Codification: What are good examples of higher-order connectors? Are there common patterns for adding new capabilities to old connectors? Is there a small basis set of operators that can produce many of the common kinds of connectors and connector extensions currently in use? What implementation techniques are used to massage existing connectors into new forms?
- Notation: How can we write down higher-order connectors so we can understand and analyze them? How rich a notation is needed? Can it be declarative?
- Semantics: What kind of semantic treatment can we give connector operations? What exactly is being operated on and how? Are there a basic set of combinators out of which all other higher-order connectors can be produced?
- Algebra: What are the algebraic relationships between different connectors? Can these relationships form the basis of a "connector calculus". For example, one might imagine a rule of idempotency would apply to a monitoring higher-order connector. (That is, adding a monitor to a monitored connector does not change it.) Or, one might ask about commutativity of the operators.
- Tools: How can we provide a rich set of higher-order connectors to architectural design environments? Can we translate higher-order connectors into code modification templates? Does this substantially simplify the development of architectural compilers?

These issues remain a promising avenue for research and development that can benefit both from good formal underpinnings, as well as application to real systems.

References

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, Ontario, Canada, November 1997.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, pages 164–174, New Orleans, Louisiana, December 1994.

[SDK$^+$95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[Sha93] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.