On the Role of Connectors in Modeling and Implementing Software Architectures

Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine, CA 92697

Technical Report 98-04

February 15, 1998

Abstract

Software architectures are software system models that represent the design of a system at a high level of abstraction. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. A notable feature of many architectural models (and the languages used to express them) is their representation of communication concerns in explicit model elements, which are typically called connectors. However, there is little consensus yet in the software engineering community on the role of connectors in an architectural model, or even on the necessity of making them first-class model elements. In this paper we demonstrate the utility of explicit connectors in architectural models through a presentation and analysis of an architecture for a meeting scheduler system. We show how the functional abstraction provided by connectors contributes to the mobility, distribution and extensibility of the design, as well as its ability to sustain runtime structural change. Furthermore, we demonstrate how connectors encapsulate important aspects of inter-component communication, including the number and identity of communication recipients, the policy used to select these recipients, the choice of implementation technology for the communications, and architectural constraints on component composition.

On the Role of Connectors in Modeling and Implementing Software Architectures

Peyman Oreizy David S. Rosenblum Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
{peymano,dsr,taylor}@ics.uci.edu

Abstract

Software architectures are software system models that represent the design of a system at a high level of abstraction. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. A notable feature of many architectural models (and the languages used to express them) is their representation of communication concerns in explicit model elements, which are typically called connectors. However, there is little consensus yet in the software engineering community on the role of connectors in an architectural model, or even on the necessity of making them first-class model elements. In this paper we demonstrate the utility of explicit connectors in architectural models through a presentation and analysis of an architecture for a meeting scheduler system. We show how the functional abstraction provided by connectors contributes to the mobility, distribution and extensibility of the design, as well as its ability to sustain runtime structural change. Furthermore, we demonstrate how connectors encapsulate important aspects of inter-component communication, including the number and identity of communication recipients, the policy used to select these recipients, the choice of implementation technology for the communications, and architectural constraints on component composition.

1 Introduction

Software architectures are software system models that represent the design of a system at a high level of abstraction [8,10]. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. Details at lower levels of abstraction, such as the selection of data structures and algorithms for individual modules, are the concern of later stages of design.

A notable feature of many architectural models (and the languages used to express them) is their representation of communication concerns in explicit model elements, which are typically called *connectors*. However, there is little consensus yet in the software engineering community on the role of connectors in an architectural model, or even on the necessity of making them first-class model elements.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973, by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-94-C-0218, and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research or the U.S. Government.

In this paper we demonstrate the utility of explicit connectors in architectural models through a presentation and analysis of an architecture for a meeting scheduler system. The architecture is constructed according to the C2 architectural style, whose connectors offer a number of important benefits over traditional approaches to system design [11].

2 A Traditional Design for a Meeting Scheduler

In the traditional approach to designing a meeting scheduler, the designer would first settle upon a well-known system organization (i.e., what a software architect would call an *architectural style*) and would then begin to allocate functional requirements to design elements. The design elements would be modules, subsystems, packages, subroutines—entities that the software architect would call *components*. As is common in the traditional approach, the designer would typically partition communication responsibilities among the different components, rather than defining them within an integrated design element that is separate from the component definitions.

For instance, one obvious design for the meeting scheduler is a client-server design. The initial design would typically involve a single server and several identical clients. The clients would provide a user interface through which meeting proposers and attendees can invite attendees, propose meetings, specify preference sets and exclusion sets, and specify resource requirements. The server would be responsible for *both* facilitating communication between the clients and for managing all of the information about meetings, meeting resources and availabilities.

Eventually, the basic client-server design might need to be scaled to support electronic meetings over wide-area networks. In this case, the single server would be evolved into a collection of federated servers that distribute the communications in some way and partition the information that they manage.

Another possible design for the meeting scheduler would use an event-condition-action style of interaction. This design may superficially resemble the client-server design, since transactions involving meetings, resources and availabilities would be directed to a server-like active database component. But communication would occur in a more asynchronous fashion than in the client-server design. As with the client-server design, an attempt to scale the design to a wide-area network would require redesign of the "server" component into a distributed active database.

While these designs represent reasonable solutions to the problem of distributed meeting scheduling from the viewpoint of traditional approaches to software design, they are nevertheless problematic from a number of viewpoints. First, while communication between the clients is a significant aspect of the requirements, the allocation of those requirements to design elements is constrained by the need to partition and encapsulate functionality across components. Thus, the design ends up containing no coherent, isolated representation of the communication that is to occur between components. This constraint leads then to the second problem, which is that the lack of explicit representation of communication leads to difficulties in the evolution of the system. One can easily imagine that the task of evolving a single, centralized server into a synchronized set of federated servers is a significant undertaking, and that the two server designs would have little in common and would provide limited opportunities for reuse.

3 Architectural Connectors

A *software architecture* represents software system structure at a high level of abstraction, and in a form that makes it amenable to analysis, refinement, simulation, and other engineering concerns [6]. The notion of a *connector* as an explicit architectural element is to be found in the earliest papers on software architecture [8]. The basic rationale for connectors is that they explicitly represent facilities for communication between components, which themselves represent the encapsulation of computation in a

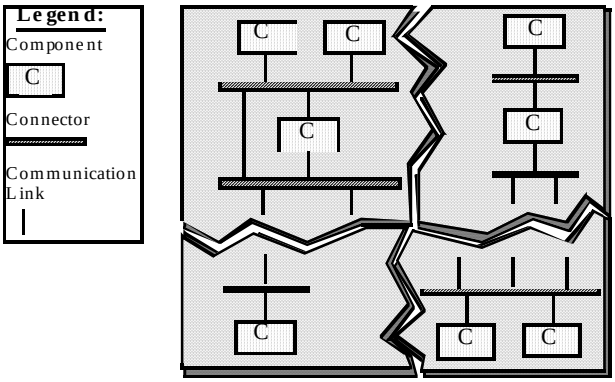

Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

system. This rationale implies that connectors should be an explicit feature of any language for modeling software architectures. Such languages are called *architecture description languages*, or ADLs, and while the notion of a connector arguably has a strong intuitive appeal, not all designers of ADLs agree on their importance as a language feature. Indeed, in many well-known ADLs, inter-component connection is specified as part of the definitions of the affected components, rather than as an explicit, separate model element. Some of the better-known examples of such ADLs include Rapide [3,4] and Darwin [5]. Medvidovic calls such connection specifications *implicit connectors* [7].

Thus, architectural connectors provide a means for separating and making explicit the communication needs of a software system, and explicit connectors can be found in many ADLs, including Wright [1] and C2 [11]. Yet in attempting to formalize the notion of connectors as language elements, the designers of ADLs have struggled to maintain a sufficient distinction between components and connectors. In describing connectors and their behavior, it is particularly easy for the description to degenerate into something that makes connectors sound little different from components—they encapsulate functionality, they can encapsulate state, they interact with other components, and so on. Perhaps the clearest way of distinguishing the two is to view components as independent units of reuse, while connectors represent the *shared phenomena* of the components they connect.[1] In other words, components encapsulate specific functionality that can be used in many different applications, and they can execute autonomously. However, connectors exist only to serve the interaction needs of components. Furthermore, connectors do need to be just a modeling abstraction; they also provide benefit when they are explicit entities in the implementation.

In most ADLs that support explicit connectors, such as Wright, the connectors are defined with a finite and statically specified number of *ports*, or interaction points for components. This is perhaps one reason they resemble components so much, since the usual method of defining a component is to specify the contents of its interface, which typically contains a finite, predetermined number of interface elements (attributes, operations, exceptions, etc.). C2 supports a much different style of connector, which can best be understood in terms of the C2 architectural style.[2]

A C2 architecture is a hierarchical network of concurrent components linked together by connectors in accordance with a set of style rules. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a connector (see Fig. 1). The

[1] Jackson describes the notion of shared phenomena, which in this context technically refers to the shared phenomena of the *domains* of the components that a connector connects [2].

[2] For a more detailed discussion of the C2-style and its benefits see [11].

style does not place restrictions on the implementation language or granularity of the components. It does require that all communication between components occur by exchanging asynchronous messages through connectors. Since all message passing is done asynchronously, control integration issues are greatly simplified[3]. This remedies some of the problems associated with integrating components that assume that they are the application's main thread of control. Furthermore, components cannot assume that they will execute in the same address space as other components or share a common thread of control.

The layering of a C2 architecture is significant in that a component is only aware of the components above it, and *explicitly* utilizes their services by sending a request message. Communication with components below occurs *implicitly*. Whenever a component changes its internal state, it announces the change by emitting a notification message, which describes the state change, to the connector below it. The connector broadcasts these notification messages to all the components connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to the state change a component.

While connectors are modeled as explicit entities in several ADLs, they are not typically retained as explicit implementation entities. Instead, they are reified as procedure calls, data accesses, or linker instructions. C2 connectors are different in that they are explicit, runtime entities in the implementation responsible for routing messages between components. Additionally, the number of ports, or interaction points, on a C2 connector can change during runtime as components are added to and removed from it. The explicit and flexible nature of C2 connectors directly contributes to our ability to implement distribution, mobility, and runtime structural change. We describe these in the context of a C2-style architecture for the meeting scheduler.

4 A C2-Style Architecture for the Meeting Scheduler

In our C2-style architecture for the meeting scheduler, each user's display consists of three windows. A "meeting proposal" window allows scheduling of a new meeting. It lets the user specify the meeting agenda, select meeting attendees, and specify a meeting time and duration. A "schedule" window displays a list of the meetings the user has agreed to attend. An "invitation" window appears each time the user is invited to a meeting and allows the user to either accept or decline the invitation. If the user accepts the invitation, the meeting is added to the "schedule" window.

The C2-style architecture for the meeting scheduler is depicted in Fig. 2. White boxes represent components and gray boxes represent connectors. A line between a component and a connector represents a communication pathway between the two. The *Person ADT*, which manages an individual user's schedule, is created for each user in the system. Each user also has three artist components, each of which is responsible for displaying a graphical user interface for one of the three meeting scheduler windows described above. The meeting proposal artist (MP Artist) manages the "meeting proposal" window; the invitation manager artist (IM Artist) manages the "invitation" window; and the schedule manager artist (SM Artist) manages the "schedule" window. The *Local connector* routes the graphical rendering messages from the artists to the *graphics* component.

Consider a scenario in which four people are using the meeting scheduler and user 1 decides to schedule a new meeting with user 3 and user 4. User 1 uses the "meeting proposal" window managed by the *MP Artist* to specify the meeting time, location, and attendees. User 1's *MP Artist* then sends meeting invitation messages to *Person$_3$ ADT* and *Person$_4$ ADT*. Upon receiving the message, *Person$_3$ ADT* and

[3] While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components.

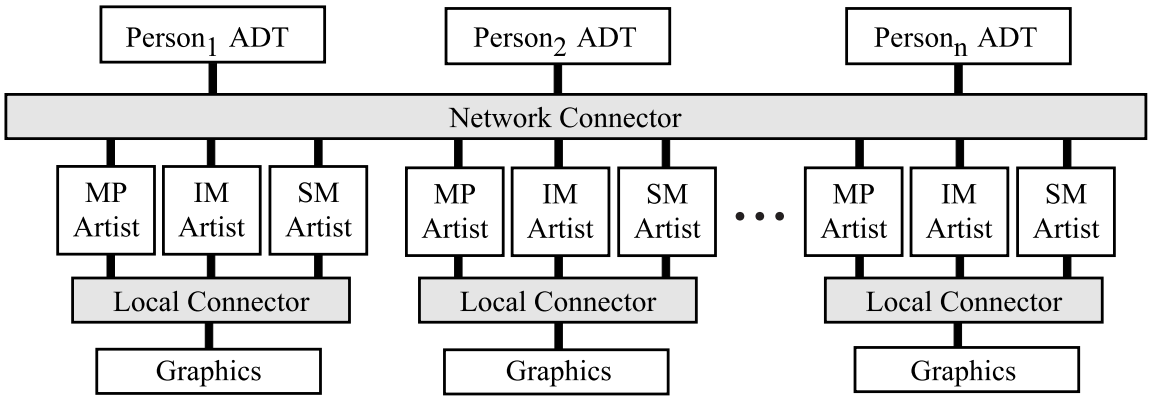

Fig. 2. The C2-style architecture for the meeting scheduler systems.

Person$_4$ ADT each record the invitation in an internal data structure and emit a state change notification message. The notification is broadcast to all the components connected below *Network Connector*. User 3 and User 4's *IM Artists* react to the notification message from their *Person ADT* by displaying a new "invitation" window. If user 3 accepts the invitation, its *IM Artist* sends a message to *Person$_3$ ADT* confirming user 3's attendance at the meeting. Upon receiving the message, *Person$_3$ ADT* updates the user's schedule and emits a state change notification. User 3's *SM Artist* reacts to this notification by updating user 3's "schedule" window; the other meeting invitees' *SM Artists* react to this notification by noting user 3's acceptance in their "schedule" windows.

The functional abstraction provided by the connector facilitates the modeling and implementation of several unique properties of our design:

- *Mobility*—The *Network connector*, which is used to broadcast notifications about new meetings, and invitation acceptances and declinations, spans both multiple users and multiple machines. For instance, the *Network connector* routes meeting invitation messages from the *IM Artist* to the *Person ADTs* of the users that have been invited. The *Network connector* is the only entity in the architecture that knows the network location of individual components.

- *Distribution*—In contrast to other meeting scheduling applications such as Microsoft's Exchange Server and Sun Microsystems' Calendar Manager, there is no centralized meeting schedule server or ADT. Instead, each user has an individual meeting schedule ADT, namely the *Person ADT*, running on their local host. This ADT only stores information about the user's scheduled meetings. We can however emulate the centralized meeting scheduler design by placing all the *Person ADTs* on a single network host and by using a common database to store schedules.

- *Runtime structural change*—The runtime addition and removal of a user does not adversely affect other users. When a user comes online, their *SM Artist* broadcasts a message to all *Person ADTs* querying for meeting invitations that it missed while offline. The notifications resulting from the query are used to synchronize the user's display with that of the other users. It should be noted that supporting this type of runtime flexibility in a completely decentralized implementation has one drawback— a user will never learn of a meeting invitation unless at least one other meeting invitee is online at the same time.

- *Extensibility*—The loose coupling between the components afforded by the connectors enables different users to have differing implementations of the components, tailored to their particular tasks and needs. This is possible as long as the different implementations adhere to the inter-component communication protocols governed by the connector.

5 Benefits Obtained from C2 Connectors

In other architectural styles, such as the client-server style, there is no coherent, isolated entity representing the communication between components[4]. As a result, decisions regarding inter-component communication are spread throughout individual application components. Connectors isolate a component's interfacing requirements from its functional requirements [9], thereby localizing decisions regarding communication policy and mechanism. C2 connectors go a step beyond other ADL connectors in that C2 connectors are explicit, runtime entities in the implementation. This enables C2 connectors to encapsulate:

1. the identity of the component receiving a particular message;
2. the number of components receiving a particular message;
3. the policy used to determine which components (from a set of eligible components) receive a message—If two or more components on a connector provide similar functionality, the connector may determine the most appropriate component to receive a given message. The decision may be based on communication latency, machine load, etc.;
4. the particular inter-process communication mechanism used for message passing—The connector can isolate the particular communication mechanism used to pass messages from one component to another (e.g., direct procedure calls, UNIX sockets, RPC, DCOM, CORBA);
5. the component's location in the network—Since components are not statically bound to one another, a component may migrate from one network node to another without having to notify other components;
6. the mapping from messages sent to message received—Since the connector acts as a conduit for communication, it can act as a domain translator between components;
7. the particular packaging and middleware technology used to implement components—Several different component packaging and middleware technologies exist for exposing the functionality of a component in a standard way. Connectors have the potential to act as a bridge between different technologies. Popular formats include COM, CORBA, Windows DLLs, and compiled Java byte-codes. The component packaging and middleware technology standardizes such things as how a component's methods are exposed for use; the invocation mechanism such as procedure calls, callbacks, and event loops; the proper order and types for passing method parameters; and the component's binary representation on disk. As long as a component adheres to the standard, the particular implementation language used by the component is inconsequential; and
8. the message to method mapping—If a component does not process C2 messages directly, the connector can provide a message to method mapping. This mapping, like the dynamic dispatch mechanism in Lisp, can potentially be altered during runtime. In fact, the binding does not have to be one-to-one. The connector may map a single message to several methods and combine the results in an appropriate manner.

As a result, architectural issues concerning these items may be considered separately from component functionality. This simplifies the behavioral model of components and enables us to more effectively partition the space of design issues.

[4] While one could attempt to model inter-component communication of such systems at the level of the computer network, the amount of detail and the potential discontinuity between the architectural model and the network topology obfuscates rather than elucidates the interactions.

We have successfully built C2 connectors that support items 1, 2, 4, and 5. Our C2 connectors also allow items 1 and 2 to be changed at runtime, enabling us to alter a system's structure during runtime. We are currently in the process of implementing a C2 connector that also supports item 7.

6 Discussion and Conclusion

In this paper, we have demonstrated the benefits of explicit connectors in architectural models. Connectors contribute to a separation of concerns in architectural modeling—they provide a convenient way to separate issues concerning component behavior from component interaction. This is especially important when constructing systems from reusable off-the-shelf components, since the designers of those components cannot anticipate all of the contexts in which the component is used. Our experience also demonstrates the utility of retaining the connectors in the implementation. The resulting functional abstraction contributes to the mobility, distribution and extensibility of the meeting scheduler, as well as facilitating runtime structural changes.

Finally, connectors provide a natural place for representing, analyzing and enforcing *architectural constraints*, which is a research topic of much current interest. In systems built from independently-constructed, off-the-shelf components, it is necessary to enforce constraints on how components are connected together and how they interact with each other. Furthermore, in many cases, it is most appropriate to evaluate and enforce such constraints at runtime. For instance, in the meeting scheduler architecture of Fig. 2, consider a Local Connector and the interactions it facilitates between the Artist components and the Graphics component. It is desirable to ensure that there is no overlap or duplication in the notifications that the Artist components send to the Graphics components, since otherwise the display presented by the Graphics component may become corrupted. The appropriate place to express and enforce this constraint is at the Local Connector, since the constraint applies to the set of components connected by the connector and not to any single component. A constraint is thus an invariant on the connector; it applies even when individual components are added to or removed from the connector. For example, the overlap constraint described above would prevent a newly added Agenda Artist from adversely affecting the user's display. If the constraints were associated instead with the Artists, then the constraints would have to be re-specified every time such a change is made to the configuration. Connection constraints such as these may be derivable from component interface specifications, but they still need to be evaluated and enforced in the context in which the components are used.

Acknowledgements

The C2-style architecture of the meeting scheduler described here is based on the one originally designed and implemented by Deborah L. Dubrow.

References

[1] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

[2] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*: ACM Press/Addison-Wesley, 1995.

[3] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–355, 1995.

[4] D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, 1995.

[5] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

[6] N. Medvidovic and D.S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages", *Proc. USENIX Conference on Domain Specific Languages*, Santa Barbara, CA, pp. 199–212, 1997.

[7] N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, 1997.

[8] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[9] J.M. Purtilo, "The POLYLITH Software Bus", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151–174, 1994.

[10] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

[11] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.