Software Connectors: A Taxonomy Approach

Nikunj R. Mehta

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
mehta@usc.edu

1 OVERVIEW

Software systems of today are frequently composed from prefabricated, heterogeneous components that provide complex functionality and engage in complex interactions. Current approaches to address the problem of consistently engineering large, complex software systems. are centered around composing software systems from coarse-grained *components*. Component-based development mostly focuses on component structure, interfaces, and functionality. An important aspect, particularly magnified by the emergence of the Internet and the growing need for distribution, is *interaction* among components. Component interaction is embodied in the notion of *software connectors*. Connectors manifest themselves in a software system as shared variable accesses, table entries, buffers, instructions to a linker, procedure calls, networking protocols, pipes, SQL links between a database and an application, and so forth [9]. In large, and especially distributed systems, connectors become key determinants of system properties, such as performance, resource utilization, global rates of flow, scalability, reliability, security, evolvability, and so forth.

Despite this critical role of connectors and recurring calls for their explicit treatment [1,2,7,8], the large-scale-development efforts have failed to adequately address and exploit them. This has resulted in their inconsistent treatment and a notable lack of understanding of what the fundamental building blocks of software interaction are and how they can be composed into more complex interactions. Middleware packages provide a predefined set of software interaction capabilities that cannot be easily extended. Also, both component-based and middleware technologies assume a homogeneous environment in which all components adhere to certain design, implementation, packaging, and runtime constraints, further aiding their prescribed types of interaction. Software architectures, on the other hand, do not assume component homogeneity, nor do they constrain the allowed connectors and connector implementation mechanisms.

With the prevailing level of understanding and support for connectors in architectures, there is an inconsistent treatment of connectors and sometimes contradictory assumptions are made. For example, connectors are often considered to be explicit at the level of architecture, but intangible in a system's implementation. This belief has at least partly contributed to the existing lack of understanding of the relationship between high-level and implementation-level connectors [7, 3]. None of these approaches furthers our understanding of what the fundamental building blocks of software interaction are and how they can be composed into more complex interactions. We have conducted an extensive study and classification of interaction mechanisms employed in software systems. resulting in a taxonomical representation of software connectors. The classification supports deeper understanding of existing connectors and their relationships. It also provides the information needed to design new, more powerful connectors by combining existing mechanisms.

2 CLASSIFICATION FRAMEWORK AND TAXONOMY

Software components perform computations and store the information relevant to an application domain; software connectors, on the other hand, perform the transfer of control and data among components. Connectors can also provide services, such as persistence, invocation, messaging, and transactions, that are largely independent of the interacting components' functionality. Capturing these facilities as connectors helps simplify an architecture and keep the architectural focus on domain-specific information. Treating these services as connectors rather than components can also help their reuse across domains.

Our connector classification framework is based on the following definition of connectors [9]:

> *Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.*

The underlying, elementary building blocks of every connector are the primitives for changing the processor program counter (control transfer) and performing memory access (data transfer). These primitives give enough conceptual power to build sophisticated and complex connectors. In addition to these primitives, every connector maintains one or more *ducts*, which are used to link the interacting components and support the flow of data and control between them. A duct is necessary for realizing a connector, but by itself, it does not provide any additional interaction services. Very simple connectors, such as module linkage, provide their service simply by forming ducts between components. Other connectors augment ducts with some combination of data and control flow to provide richer interaction services. Connectors can also have an internal architecture that includes computation and information storage. For example, a load balancing connector would execute an algorithm for switching incoming traffic among a set of components based on the knowledge about the current and past load state of components.

Simple connectors are typically implemented in programming languages. On the other hand, composite connectors are achieved through composition of several connectors (and possibly components), and are usually provided as libraries and frameworks. Simple connectors only provide one type of interaction services, whereas composite connectors may combine many kinds of interactions. However, when creating such connectors, it is important to have a conceptual framework of reasoning about their underlying, low-level interaction mechanisms. A connector taxonomy realizes this conceptual

framework, provides a mechanism for identifying design choices and detecting architectural mismatches, and can serve as a tool for architectural composition.

Each connector is identified by its primary service category and further refined based on the choices made to realize these services. The characteristics most commonly observed among connectors are positioned towards the top of the framework, whereas the variations are located in the lower layers. The framework comprises service categories, connector types, dimensions, sub-dimensions, and values for the dimensions. A *service category* represents the broad interaction role the connector fulfills. Connector *types* discriminate among connectors based on the way in which the interaction services are realized. The architecturally relevant details of each connector type are captured through *dimensions*, and, possibly, further *sub-dimensions*. Finally, the lowest layer in the framework is formed by the set of *values* a dimension (or sub-dimension) can take. Note that our classification does not result in a strict hierarchy, but rather in a directed acyclic graph (DAG). Species of connectors are created by choosing the appropriate dimensions from connector types. It is possible to combine dimensions in an arbitrary fashion although some of these would be infeasible.

Figure 1 shows the taxonomy where the interaction services are shown on the extreme left, the species on the extreme right, and connector types, dimensions, sub-dimensions, and values in between. Occasionally, species are highlighted against a single relevant dimension for illustration, although they would also have values for other dimensions.

Service Categories

The topmost layer in our classification framework is the service category, which specifies the interaction services a connector provides. We identify the following four categories of interaction services as a refinement of the connector roles described by Perry [5]. The categories fully describe the range of possible component interactions:

Communication. Communication connectors support transmission of data among components. Data transfer services are a primary building block of component interaction.

Coordination. Coordination connectors support transfer of control among components. Components interact by passing the thread of execution to each other.

Conversion. These connectors convert the interaction required by one component to that provided by another. Enabling heterogeneous components to interact with each other is a non-trivial task due to possible interaction mismatches.

Facilitation. Facilitation connectors mediate and streamline component interaction. Even when heterogeneous components have been designed to interoperate with each other, there is a need to provide mechanisms for facilitating and optimizing their interactions.

Every connector provides services that belong to at least one of these categories. It is also possible to have multi-category connectors to satisfy the need for a richer set of interaction services. For example, it is possible to have a connector that provides both communication and coordination services.

Connector Types, Dimensions, and Values

Interaction services can be used to perform a broad categorization of connectors, but this leaves a lot of details unexplained. This level of abstraction cannot help us build new connectors, nor can it be used to model and analyze them (e.g., in an architecture). Hence, we further classify connectors into different types based on the way in which they realize interaction services: procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor. Connector types are the level at which architects typically consider interactions when modeling systems.

Simple connectors can be modeled at the level of connector types; their details can often be left to design and implementation. On the other hand, more complex connectors often require that many of their details be decided at the architectural level so that the impact of these decisions can be studied early and on a system-wide scale. Those details represent variations in connector instances and are treated as connector dimensions in our taxonomy. In turn, each dimension has a set of possible values. The selection of a single value from each dimension results in a concrete connector species. Instantiating dimensions of a single connector type forms simple connectors; on the other hand, using dimensions from different connector types leads to a composite (“higher-order”) connector species.

3 FURTHER WORK

We do not hypothesize that the connector taxonomy is complete in its current form. Instead, it is intended to enable better and more complete understanding of software connectors and to evolve as that understanding improves. In particular, we feel that our recognition that every connector comprises a set of ducts and engages in transfers of data and/or control will remain valid. We also believe the four connector services and eight types to be fairly stable. During the process of creating the taxonomy and evaluating it on numerous examples over the past year, we have found that all encountered connectors could be classified as providing one or more of the services and belonging to one (in the case of simple connectors) or more (in the case of higher-order connectors) of the types. On the other hand, the dimensions, sub-dimensions, and values are likely to evolve as our understanding of connectors evolves.

Creating unprecedented connectors is not a trivial task. Just as component integration has presented tremendous challenges to software engineers, so too will the integration of heterogeneous connectors. Much work is required to identify the nature of dimension compatibility and orthogonality. Certain dimensions can also influence the use of other dimensions. A taxonomy such as the one we propose is a necessary precursor to identifying the relationships among connector types, their dimensions, and values.

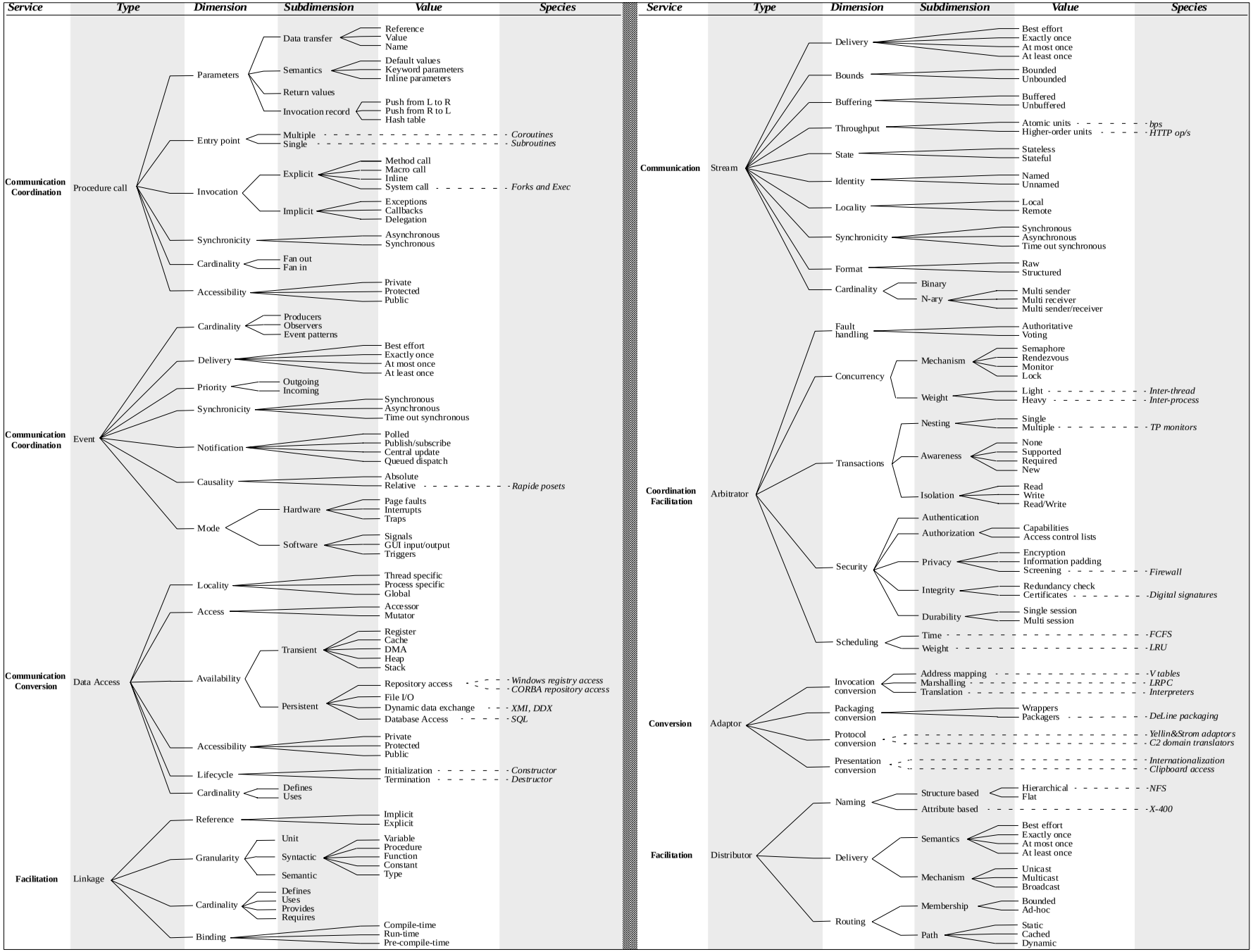

Figure 1. Connector Taxonomy [4]. A small number of connector species is included for illustration purposes.

Many issues remain venues of future work. We intend to further study connector properties, relationships, and tradeoffs in order to devise novel connectors to help automate programming tasks that currently require manual, but recurring solutions. For example, we intend to investigate the possibility of constructing a parallel execution connector that would allow developers to eliminate application-specific constructs for parallel execution from components. We will also study the possibility of automatically adding distribution support (via a distributor connector) to connectors that only support local interactions.

We will investigate these and similar problems in the context of a toolset that will allow experimentation with and evaluation of connectors. An initial prototype of the toolset, built in Java, is already operational. Thus far, in addition to the simple connector types provided by Java (procedure calls and data access), we have begun exploring event connectors. This work is an extension of our previous work with message-based connectors used in the context of the C2 architectural style [10]. A planned, future extension of the toolset will allow us to measure and monitor execution characteristics of connectors to enable their runtime augmentation (e.g., to support better load balancing) and replacement. Finally, we intend to use this work as a platform for studying the trade-off between connector efficiency and adaptability, identified as the key factor in effectively supporting architecture-based runtime software evolution [6].

4 REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
3. D. Hirsch, S. Uchitel and D. Yankelovich. Towards a Periodic Table of Connectors. In *Proceedings of the Symposium on Software Technology* , 1999. Buenos Aires, Argentina.
4. N. R. Mehta, N. Medvidovic and S. Phadke. Towards a Taxonomy of Software Connectors, In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000 (to appear).
5. D. E. Perry. Software Architecture and its Relevance to Soft-ware Engineering, Invited Talk. *Second International Conference on Coordination Models and Languages,* Berlin, Germany, September 1997.
6. R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 6(1), November 1986.
7. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. *Workshop on Studies of Software Design*, 1993.
8. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.
9. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
10. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. White-head, J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transaction on Software Engineering*, 22(6), 1996.