Towards a Taxonomy of Software Connectors

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

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

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

ABSTRACT

Software systems of today are frequently composed from prefabricated, heterogeneous components that provide complex functionality and engage in complex interactions. Existing research on component-based development has mostly focused on component structure, interfaces, and functionality. Recently, software architecture has emerged as an area that also places significant importance on component interactions, embodied in the notion of software connectors. However, the current level of understanding and support for connectors has been insufficient. 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. This paper attempts to address this problem. It presents a comprehensive classification framework and taxonomy of software connectors. The taxonomy is used both to understand existing software connectors and to suggest new, unprecedented connectors. We demonstrate the use of the taxonomy on the architecture of an existing, large system.

Keywords: software architecture, software connector, classification, taxonomy

1 INTRODUCTION

A number of techniques has recently emerged to address the problem of consistently engineering large, complex software systems. The three most widely embraced efforts have been component-based software development standards (e.g., [5,18]), middleware (or software interoperability) platforms (e.g., [28,38]), and software architecture [31,41]. They present complementary, often overlapping approaches, centered around composing software systems from coarse-grained *components*.

Although components have been the predominant focus of researchers and practitioners, they address only one aspect of large-scale development. Another 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 [41]. 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 [1,10,39,40], the large-scale-development efforts discussed above have failed to adequately address and exploit them. The primary concern of component-based approaches is functionality. Component interactions play a secondary role: the interaction details are entirely hidden inside individual components. Middleware approaches primarily focus on the infrastructure necessary to enable components to interact. A middleware package provides a predefined set of software interaction capabilities that is not intended to be extensible. Additionally, 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 architecture-based approaches have come furthest in their treatment of connectors. They typically separate computation (components) from interaction (connectors) in a system. In principle, architectures do not assume component homogeneity, nor do they constrain the allowed connectors and connector implementation mechanisms. Several existing architecture-based technologies have provided support for modeling or implementing certain classes of connectors [1,40, 45].

Despite this, the current level of understanding and support for connectors in architectures is insufficient. With some exceptions [1,30,39], the architecture community has thus far maintained a studied silence on the exact nature of connectors. This has resulted in their inconsistent treatment and sometimes contradictory assumptions. 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 [39]. Connectors are also sometimes deliberately modeled as components (e.g., the notion of a "connection component" in Rapide [22]), further obscuring their distinct nature. Those architectural approaches that have explicitly addressed connectors have either provided mechanisms for *modeling* arbitrarily complex connectors or *implementing* a small set of simple ones, but never both [10].

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. Such level of understanding is necessary in order to fully meet the challenge posed to software architecture researchers of giving connectors “a first-class status” [23,39]. This paper takes a step in that direction. The paper presents an extensive study and classification of interaction mechanisms employed in software systems. 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. The ultimate goal of this work is to produce a comprehensive taxonomy of software connectors; the classification presented in this paper forms the foundation for this endeavor.

Our classification builds upon three atomic elements of software interaction: all software connectors comprise one or more *ducts*, interaction channels with no associated behavior; furthermore, all connectors, regardless of their complexity, provide mechanisms for *transferring data* and/or *control* along a duct. The classification identifies four major categories of connectors, based on the *services* they provide to interacting components: *communication*, *coordination*, *conversion*, and *facilitation*. We also identify the major *connector types*: *procedure call*, *data access*, *linkage*, *stream*, *event*, *arbitrator*, *adaptor*, and *distributor*. Each connector type supports one or more interaction services. We enumerate *dimensions* of variation within each type, and possible *values* for each dimension. We use an extended example to illustrate how the resulting classification framework can be used to better understand an existing system. In the course of doing so, we highlight some interesting relationships among connector types.

The remainder of the paper is organized as follows. To motivate the classification framework, Section 2 presents a brief example of decomposing two different, high-level connectors into overlapping sets of lower-level interaction primitives. Section 3 discusses the work of other researchers on understanding and classifying connectors. Section 4 presents the classification framework, while Section 5 shows a more extensive example of identifying the different types of connectors in an existing, large system. The paper concludes with a discussion of lessons learned and a brief overview of future work.

2 MOTIVATING EXAMPLE

Different views of a software connector are useful for different tasks. In order to model a system and communicate its properties, a high-level view is suitable. For example, an architect may make the following concise, but meaningful statement about the configuration shown in Figure 1a: “Components A and B communicate via a Unix *pipe*.” That statement may be accompanied by a formal specification of the pipe’s overall behavior (e.g., [1,34]). However, such a high-level description does not help one understand all the properties of the pipe, how it can be adapted, or under what conditions it can be replaced with another type of connector. A more detailed, lower-level view is needed to accommodate that.

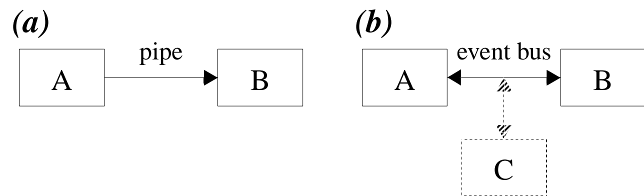

Figure 1. ***(a)*** A Unix pipe supports unidirectional component communication. ***(b)*** An event bus supports bidirectional communication and addition/removal of components.

In particular, the pipe in Figure 1a allows interaction via unformatted streams of data. It is a simple connector: it consists of a single duct and facilitates only unidirectional data transfer. Thus, the cardinality of the pipe is a single sender and a single receiver. The pipe allows components A and B to exhibit very low coupling: the components do not possess any knowledge about one another. For example, A’s task is only to successfully hand off its data to the pipe; the actual recipient (if any) is unimportant to A. In turn, the pipe in Figure 1a does not buffer the data. It attempts to deliver the data at most once; if the recipient is unable to receive the data for some reason, the data will be lost.

Let us now assume that we need to alter the manner in which A and B interact, such that B can also send information (e.g., acknowledgement of data receipt) back to A. Furthermore, we wish to ensure the delivery of data: if the recipient is not available, the pipe retries to send until the data is successfully transferred. Both these modifications can be potentially accommodated by pipes. The first would require introducing another pipe from B to A, and the second, data buffering in a pipe. Addition of any other components into the system would require further addition and/or replacement of pipes (possibly requiring substantial system down-time). Pipes can, therefore, accommodate these new requirements, even though constantly adding and replacing them may not be the most effective solution.

If, however, we want to change the nature of data from an unformatted stream to discrete, typed packets that can be processed more efficiently by interacting components, pipes will not suffice. It is in such a case that a taxonomy of software connectors, such as the one we propose, can help in determining a suitable alternative. For example, the taxonomy may suggest that all of the above requirements can be satisfied by an event bus. Although clearly different types of connectors, pipes and event buses exhibit a number of similar properties (e.g., loose component coupling, asynchronous communication, possible data buffering). At the same time, event buses are better suited to support system adaptation: a bus is capable of establishing ducts between interacting components “on the fly;” its cardinality is a single event sender (similarly to the pipe), but multiple observers. Thus, in principle, event buses allow components to be added or removed, and to subscribe to receive certain events, at any time during a system’s execution [6,33]. Figure 1b depicts using an event bus to accommodate the above changes.

3 OVERVIEW OF RELATED WORK

Software connectors have been suggested as first-class

elements of software architectures [39]. Various architecture description languages (ADLs) provide formalisms for *modeling connectors* and their characteristics. Furthermore, *middleware* technologies provide infrastructure that implements a set of standardized connectors. Previous research has studied the influence of connectors (in the context of middleware) on software architectures and *architectural styles* [14]. Finally, the area of *distributed systems* has produced new forms of connectors to support sophisticated component interactions. These four areas form the underpinnings of our work and are further discussed below.

Connector Modeling

Programming languages typically describe interactions among modules at the level of procedure call and shared data access. Special purpose ADLs have been devised to provide support for modeling more sophisticated and powerful connectors. A predominant focus of existing ADLs has been on verifying the properties of modeled behavior. ADLs represent the services exported by connectors, mechanisms used to implement them, interaction protocols, and constraints on connector usage and evolution [23]. They do so by adopting textual or graphical modeling constructs [11,40]. Most ADLs lend themselves well to formally representing well-understood connectors. Their suitability for designing novel connectors remains unexplored.

A number of other approaches have shed light on software interaction modeling. Perry has provided a high-level classification of the roles software connectors play in an architecture [30]. Kazman et al. have suggested external canonical features for describing architectural elements, including connectors [20]. Finally, earlier attempts have modeled connectors at the level of module interdependencies [13,32]. As we will discuss further in Section 5, basing representations of system interactions on module interdependencies alone tends to hamper system understanding. Overall, the current literature lacks an understanding of how various connector characteristics are related and does not provide sufficient guidance to architects for choosing interaction mechanisms.

Middleware

As already discussed, simple connectors such as memory access and procedure call are supported by programming languages; some also provide native thread management. More sophisticated connectors are now being discussed and their support is being studied through frameworks and higher-level programming languages. Such connectors include pipes, remote procedure calls, schedulers [11,40], adaptors [46], packagers [12], active interfaces [19], fault tolerant arbiters [8], events [6,36], and real time filters [35]. The need for supporting complex components and their interactions forces one to think about the possible variations of connectors and the way in which sophisticated connectors are composed.

Operating systems and off-the-shelf (OTS) middleware platforms provide abstractions and mechanisms for supporting complex component interactions. Various research and industrial middleware platforms have emerged (e.g., JEDI [9], CORBA [28], COM [37], Enterprise Java Beans [43]). Each focuses on a particular form of component interactions such as RPC, distributed transactions, and security. However, each middleware technology supports only a small set of interaction primitives that may not be universally applicable [10].

Architectural Styles

Since connectors have a high potential for cross-application and cross-domain reuse, several architectural styles based on connectors have been identified [41]. An architectural style defines a set of rules that describe or constrain the structure of architectures and the way in which their components interact. The styles motivated by software connectors include pipe and filter [41], real-time data feeds [35], event-driven architecture [6], message-based style [45] and dataflow style [41]. Each style allows considerable flexibility in the choice of implementation of the connectors, which requires identification of the possible parameters of variation.

Distributed Systems

Distributed systems have demanding dependability constraints such as performance, safety, security, and scalability, and require that the systems adapt to changing environments [36]. Distribution also entails issues such as concurrency control, transactions, and reliability of component interactions. Distributed systems research provides the theoretical constructs for dealing with such issues [7]. Our work has tried to leverage the results of research in distributed systems to identify connectors that facilitate the desired extra-functional properties, such as those discussed above. In the process, we hope to enrich the body of dependability constructs and mechanisms available to the researchers and developers of distributed systems.

4 THE CLASSIFICATION FRAMEWORK

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' functionalities. These services are considered to be facilities *components* in middleware standards such as CORBA, COM and RMI [28,38,44]. 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 also helps their reuse across domains.

Most component-based software development approaches assume that all interactions among components consist of procedure calls and shared data access. On the other hand, middleware-based approaches assume that all component interactions take place via message passing and remote procedure calls (RPC). Both approaches thus embrace a rather narrow view of connectors that is unlikely to be successfully applicable across all development situations.

This problem is further exacerbated by the increased emphasis on development using large, OTS components originating from multiple sources. As components become

more complex and heterogeneous, the interactions among them become more critical determinants of system properties. Studies have shown that the integration of components with mismatched assumptions about their environment is a difficult problem [16,17]. It is, then, the task of connectors to mitigate such mismatches. It is in the context of these issues that we study the nature and role of software connectors.

Our connector classification framework is based on the following definition of connectors [41]:

> *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 and performing memory access. 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. Complex connectors can help us in overcoming the limitations of modern programming languages and in realizing the potential of programming-in-the-large [13]. However, when creating such connectors, it is important to have a conceptual framework of reasoning about their underlying, low-level interaction mechanisms. The taxonomy presented in this paper realizes this conceptual framework, provides a mechanism for identifying design choices and detecting architectural mismatches, and can serve as a tool for architectural composition.

The overall structure of our proposed connector classification framework is depicted in Figure 2: 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, subdimensions, 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 *subdimensions*. Finally, the lowest layer in the framework is formed by the set of *values* a dimension (or subdimension) can take. Note that our classification does not result in a strict hierarchy, but rather in a directed acyclic graph (DAG).

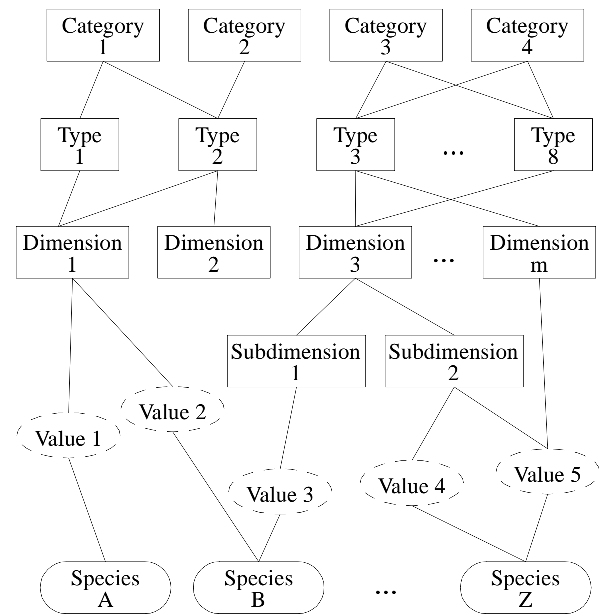

Figure 2. Structure of the connector classification framework.

The remainder of this section describes in more detail the classification framework and a comprehensive taxonomy that can be used as the foundation for classifying software connectors. Figure 3 shows the complete taxonomy and is used as the basis of discussion. The interaction services are shown on the extreme left, the species on the extreme right, and connector types, dimensions, subdimensions, 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. The categories fully describe the range of possible software component interactions:

Communication. Communication connectors support transmission of data among components. Data transfer services are a primary building block of component interaction. Components routinely pass messages, exchange data to be processed, and communicate results of computations.

Coordination. Coordination connectors support transfer of

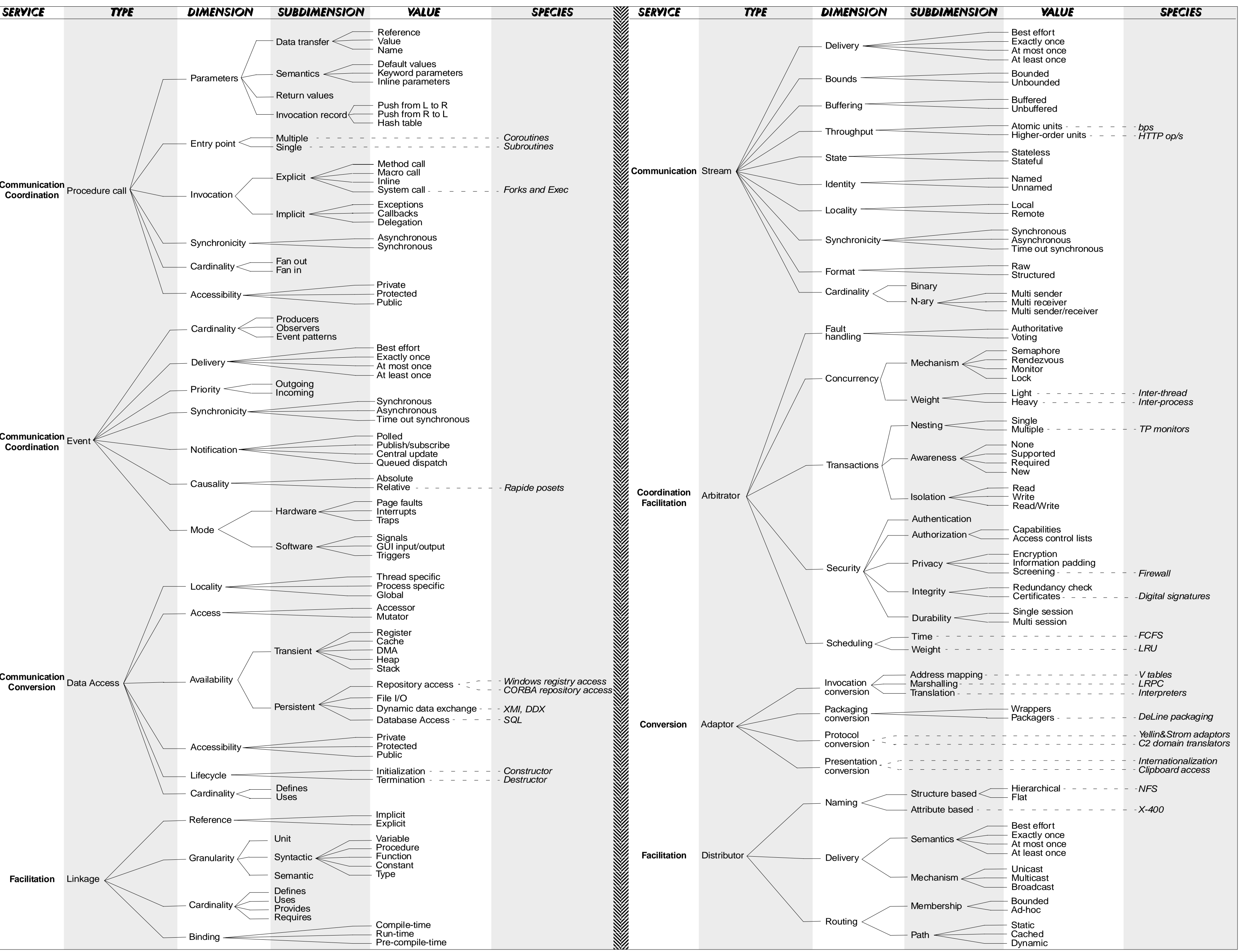

Figure 3. Connector Taxonomy.

control among components. Components interact by passing the thread of execution to each other. Function calls and method invocations are examples of coordination connectors. Higher-level connectors, such as signals and load balancing connectors, provide richer, more complex interactions built around coordination services.

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. Interaction mismatches are a major hindrance in composing large systems [16,17]. The mismatches are caused by incompatible assumptions made by components about the type, number, frequency, and order of interactions in which they are to engage with other components. Conversion services allow components that have not been specifically tailored for each other to establish and conduct interactions. Conversion of data formats and wrappers for legacy components are examples of connectors providing this interaction service.

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. Mechanisms like load balancing, scheduling services, and concurrency control are required to meet certain extra-functional system requirements and to reduce coupling between components.

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. These connector types are the interactions that architects consider 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 connector species.

Each connector type is described in more detail below. Note that our taxonomy does not define rules for composing connectors. Instead, it provides a basis for exploration of new connector species.

Procedure Call

Procedure call connectors model the flow of control among components through various invocation techniques (*coordination*). They also perform transfer of data among the interacting components through the use of parameters (*communication*). These connectors are among the best understood connectors and have been likened to the assembly language of software interconnection [39]. Examples of procedure call connectors include object-oriented methods; fork and exec in Unix-like environments, call back invocation in event-based systems, and operating system calls. Procedure calls are used as the basis for composite connectors, such as RPC [3], which also performs facilitation services.

Event

An event can be defined as “the instantaneous effect of the (normal or abnormal) termination of the invocation of an operation on an object, and it occurs at that object's location” [36]. Event connectors are similar to procedure call connectors in that they model the flow of control among components (*coordination*). In this case, the flow is precipitated with an event. Once the event connector learns about the occurrence of an event, it generates messages for all interested parties and yields control to the components for processing these messages. Messages can be generated upon the occurrence of a single event or a specific pattern of events. The contents of an event can be structured to contain more information about the event, such as the time and place of occurrence, and other application-specific information (*communication*). Virtual connectors are formed between components interested in the same event topics. Event-based distributed systems rely on the notion of time and ordering of actions [21]. Therefore, dimensions such as causality, atomicity, and synchronicity play a critical role in event connector mechanisms. Event connectors are found in distributed applications that require asynchronous communication. An example is a windowing application (such as X Windows [37]), where GUI inputs serve as the events that activate the system. Finally, some events, such as interrupts, page faults, and traps, are triggered by hardware and then processed by software. These events may affect global system properties, making it important to capture them in software architectures.

Data Access

Data access connectors allow components to access data maintained by a data store component [31] (*communication*). Data access often requires preparation of the data store before and clean-up after access has been completed. In case there is a difference in the format of the required data and the format in which data is stored and provided, data access connectors may perform translation of the information being accessed (*conversion*). The data can be stored either persistently or temporarily form, in which the data access mechanisms will vary. Examples of persistent data access include query mechanisms, such as SQL for database access,

and accessing information in repositories, such as the CORBA interface repository. Example of transient data access includes heap and stack memory access, and information caching.

Linkage

Linkage connectors are used to tie the system components together and hold them in such a state during their operation. Linkage connectors enable the establishment of ducts, the channels for communication and coordination, which are then used by higher-order connectors to enforce interaction semantics (*facilitation*).

Once ducts are established, a linkage connector may disappear from the system or remain in place to assist in the system's evolution. Examples of linkage connectors are the links between components and buses in a C2 style architecture [45] and dependency relationships among software modules described by module interconnection languages (MIL) [32].

Note that linkage connectors do not enhance the functionality of a system, but they are required to grow, monitor, and repair existing systems. Architectural description that shows interactions based on linkage connectors alone can obfuscate system understanding. An example of such an architecture is the Linux architecture as described in [4] and discussed in Section 5 below.

Stream

Streams are used to perform transfers of large amounts of data between autonomous processes (*communication*). Streams are also used in client-server systems with data transfer protocols to deliver results of computation. Streams have been employed in formal architectural models to represent connectors with fairly complex protocols of usage [1,34]. Streams are a good example of a complex connector with its own internal architecture. Various dimensions, as shown in Figure 3, allow specifying the design choices for realizing a stream connector species. Streams can be combined with other connector types, such as data access connectors, to provide composite connectors for performing database and file storage access, and with event connectors to multiplex the delivery of a large number of events. Examples of stream connectors are UNIX pipes, TCP/UDP communication sockets, and proprietary client-server protocols.

Arbitrator

When components are aware of the presence of other components but cannot make assumptions about their needs and state, arbitrators streamline system operation and resolve any conflicts (*facilitation*), and redirect the flow of control (*coordination*). For example, multi-threaded systems that require shared memory access use synchronization and concurrency control to guarantee consistency and atomicity of operations. Arbitrators can provide facilities to negotiate service levels and mediate interactions requiring guarantees for isolation levels, reliability, and atomicity. They also provide scheduling and load balancing services. Arbitrators can ensure system trustworthiness by providing crucial support for dependability [42] in the form of reliability, safety, and security.

Adaptor

Adaptor connectors provide facilities to support interaction between components that have not been designed to interoperate. Adaptors involve matching communication policies and interaction protocols among components (*conversion*). These connectors are necessary for interoperation of components in heterogeneous environments, such as different programming languages or computing platforms. Conversion can also be performed to optimize component interactions for its execution environment (e.g., LRPC [2]). Adaptors may employ transformations (e.g., table look-ups) to match required services to the available facilities.

Examples of adaptors include virtual memory translation; Yellin and Strom's adaptors [46], which match incompatible interaction protocols; virtual function tables used for dynamic dispatch of polymorphic method calls [15]; and DeLine's packagers [12]. XML metadata interchange (XMI) is a recent approach that supports interchange of models between applications and performs the data presentation conversion [26].

Distributor

Distributor connectors perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths (*facilitation*). They never exist by themselves, but provide assistance to other connectors such as streams or procedure calls. Distributed systems exchange information using distributor connectors to direct the data flow. Distributed systems require identification of component location and the paths to them based on symbolic names. Domain name service (DNS) [24], routing, switching and many other network services belong to this connector type. Distributors have an important effect on system scalability and survivability.

5 TAXONOMY IN ACTION

The objective of our taxonomy of connectors is to provide enough information about connectors to enable their treatment as first-class elements of an architecture. This would, in turn, lead to better design of complex systems. One of the risks with representing a complex system is that its architecture may become an almost fully connected graph of components (e.g., [11]). We believe this problem is typically a consequence of the use of primitive connectors—linkages. In order to test the utility of our taxonomy and the rich connectors it allows, we chose to study a complex system and assess the benefits of using more sophisticated connectors. We selected Linux as a representative example. We believe it is a good candidate also because its architecture has been recently studied [4]. Bowman et al. identify interactions among Linux components at the level of module dependencies, which are equivalent to the ducts in our framework. Ducts are not rich constructs and do not meaningfully describe component interactions. We believe that the Linux architecture can be rendered more meaningful by explicating higher-order connectors. We identified several

such connectors in the Linux architecture, three of which we discuss in this paper: the file facade, IPC via shared memory access, and the process scheduler. Each example is further discussed below.

The Linux File Facade

The file facade is a commonly used abstraction to access devices and file systems in Linux. It performs a context-dependent dispatch of incoming system calls for file operations. The actual behavior of this connector depends on the calling process and the requested resource. The facade can be used to access a variety of file systems and devices, such as an ext2 file-system, a 32-bit DOS partition, or a serial device. Consider the complexity of interactions among the calling process and the requested file resource in the case where a "file open" call is made by a process on an exported NFS volume. The file may be local or remote, and may be a link to yet another file. Further, the file may be in contention, requiring the connector to arbitrate the access. The calling process would also need to have rights to access this file, which would require performing authorization. The facade also modifies the system's state from user-mode to kernel-mode and vice-versa upon the return of the call. The facade is composed of the system call interface, the Virtual File System, the ext2 file system and the interface itself, as illustrated in Figure 4. Thus, in addition to coordination, the facade performs arbitration of interaction between the application process and physical file system and adaptation by dispatching calls to various file systems and device drivers.

Table 1: The file facade connector

Connector Type	Dimension	Value
Arbitrator	Authorization	Access Control Lists
Arbitrator	Isolation	Read/Write
Adaptor	Invocation conversion	Translation

Application of our taxonomy to this connector is shown in Table 1. Note that the facade allows simultaneous access to both readers and writers. Therefore, components making use of this connector will have to either use a mechanism to detect writers and lock the accessed resource or extend the connector to include such a mechanism.

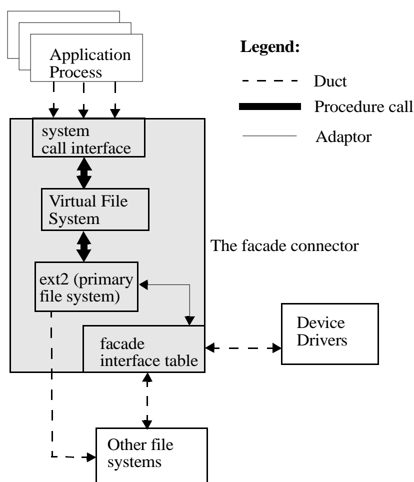

Figure 4. The facade connector.

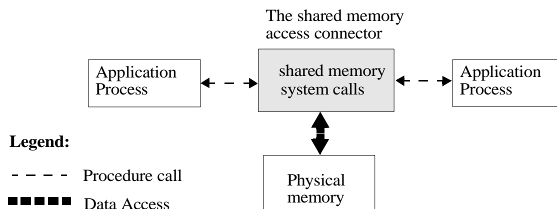

Figure 5. The shared memory access connector.

Shared Memory Access

The second connector we consider is shared memory access, a commonly used IPC mechanism on Unix-like systems. As shown in Figure 5, the shared memory access connector is structurally composed of a supporting library of system calls that allow the creation, manipulation and release of shared memory.

Applying the taxonomy to shared memory access results in Table 2. The table indicates that, as a connector, shared memory access does not facilitate arbitration of data access among multiple readers and writers. Therefore, additional connectors would be required to isolate the interactions of different components.

Table 2: Shared memory access connector

Connector Type	Dimension	Value
Data access	Locality	Global
Data access	Access	Accessor and Mutator
Data access	Transient availability	Heap
Data access	Accessibility	Protected
Data access	Cardinality	N defines; N uses

Process Scheduler

The process scheduler is responsible for switching system resources between multiple user-level processes while ensuring that undesirable system states, such as deadlocks and resource starvation, do not compromise the integrity of the system. Additionally, the scheduler must also try to optimize the utilization of system resources. We propose that the scheduler is a connector, as depicted in Figure 6. It links the operating system to the application processes, coordinates the processes' access to system resources, and abstracts the resources in a manner that allows their easy access, independently of other processes. Such an interaction is appropriately modeled as an arbitrator. Table 3 lists the relevant process scheduler dimensions and values.

Table 3: The scheduler as a connector

Connector Type	Dimension	Value
Arbitrator	Fault handling	Authoritative
Arbitrator	Concurrency weight	Heavy
Arbitrator	Authorization	Access control lists
Arbitrator	Scheduling	Time

By treating it explicitly as a connector, we allow the Linux process scheduler to be realized as an extensible entity that may be used to configure a system's characteristics. For example we may redesign its scheduling property with a prioritized queue or a cooperative multitasking algorithm that

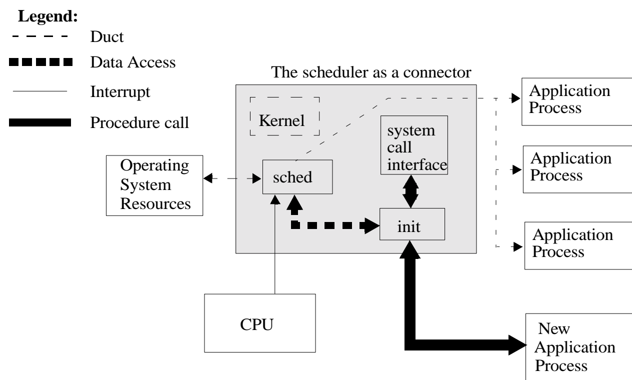

Figure 6. The process scheduler connector.

will change the system's performance accordingly.

The examples discussed in this section show the power of applying our taxonomy to a real world example. We believe that our representation of connectors better describes the component interaction semantics than Bowman et al.'s [4]. In turn, this has the potential to support richer analysis and easier understanding. Finally, the taxonomy will allow designers to select from a range of connectors in a manner that would be best suited for their system's desired extra-functional properties such as performance, extensibility, and security.

6 DISCUSSION AND CONCLUSIONS

Creating a taxonomy presents a tremendous intellectual challenge. It requires classifying the work done by hundreds of researchers over several decades into a compact framework. The main contribution of this paper is the underpinning of such a framework for charting the space of software connectors. We do not expect that our framework 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. The framework is also intended to serve as a rallying point for software engineering, and particularly software architecture researchers in realizing their shared vision of giving connectors equal status to that of components.

While we have envisioned the taxonomy to be adaptable, we do not expect that all of its elements are equally likely to change. 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, subdimensions, and values are likely to evolve as our understanding of connectors evolves.

The scope of the taxonomy is one aspect of our work that distinguishes it from other studies of connectors (e.g., [30,39]). Another unique aspect is that we do not use the taxonomy only to understand and analyze existing connectors, but also to synthesize new connector species. This was demonstrated in the context of the Linux example in Section 5. 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. At the least, it will require better understanding of the connectors' complementary and contradictory characteristics. A taxonomy such as the one we propose is a necessary precursor to identifying the relationships among connector types, their dimensions, and values.

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 [45]. 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 dynamic software evolution [27].

7 REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), 1990.
3. A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.
4. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
5. K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
6. A. Carzaniga, E. Di Nitto, D. Rosenbloom, and A. L. Wolf. Issues in Supporting Event-Based Architectural Styles. In *Proceedings of the Third International Worshop on Software Architectures*, Orlando, FL, November 1998.
7. G. Colouris, J. Dollimore and T. Kindberg. Distributed Systems: Concepts and Design. 2nd ed. Addison-Wesley, US,

1994.

8. J. E. Cook and J. A. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
9. C. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.
10. 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.
11. T. R. Dean and J. R. Cordy. A Syntactic Theory of Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.
12. R. DeLine. Avoiding Packaging Mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
13. F. DeRemer and H. H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, June 1976.
14. E. Di Nitto and D. S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
15. K. Driesen, U. Holzle and J. Vitek. Message Dispatch on Pipe-lined Processors ECOOP '95, Lecture Notes in Computer Sciences, volume 952. Springer Verlag, 1995.
16. C. Gacek and B. W. Boehm. Composing Components: How Does One Detect Potential Architectural Mismatches? *Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.
17. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995.
18. G. Hamilton, ed. JavaBeans API Specification, version 1.01. Sun Microsystems, July 1997.
19. G. T. Heineman. Adaptation and Software Architecture. In *Proceedings of the Third International Workshop on Software Architectures*, Orlando, FL, November 1998.
20. R. Kazman, P. Clements, L. Bass and G. Abowd. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *Proceedings of COMPSAC 97*, Washington, D. C., August 1997.
21. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.
22. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
23. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in *IEEE Transactions on Software Engineering*. (to appear)
24. P. Mockapetris, and K. J. Dunlap Development of the domain name system.In *Proceedings of the Symposium on Communications Architectures and Protocols*, 1988.
25. B. C. Neumann. Scale in Distributed Systems. In Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994.
26. Object Management Group. XML Metadata Interchange (XMI). Proposal to the OMG OA & DTF RFP 3: Stream-based Model Interchange Format (SMIF). OMG Document ad/98/10-05. October 1998.
27. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.
28. R. Orfali, D. Harkey, and J. Edwards. The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., NY, 1996.
29. D. E. Perry. Software Interconnection Models. *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, May 1987.
30. D. E. Perry. Software Architecture and its Relevance to Software Engineering, Invited Talk. *Second International Conference on Coordination Models and Languages*, Berlin, Germany, September 1997.
31. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.
32. R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 6(1), November 1986.
33. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4), July 1990.
34. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering*, April 1998, Kyoto, Japan.
35. N. Roodyn and W. Emmerich. An Architectural Style for Multiple Real-Time Data Feeds. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
36. D. S. Rosenblum and A. L. Wolf, A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the Sixth European Software Engineering Conference*, Zurich, Switzerland, September 1997.
37. R. Scheifler and J. Gettys, The X Window System, *ACM Transactions on Graphics*, April 1987.
38. R. Sessions. COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, Inc., NY, 1997.
39. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. *Workshop on Studies of Software Design*, 1993.
40. 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.
41. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
42. V. Stavridou and R. A. Riemenschneider. Provably Dependable Software Architectures. In *Proceedings of the Third International Worshop on Software Architectures*, Orlando, FL, November 1998.
43. Sun Microsystems, Inc. Java 2 Enterprise Edition Specification v1.2. http://java.sun.com/j2ee
44. Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com/products/jdk/rmi/index.html.
45. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, 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.
46. D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.