Formal Connectors

Robert Allen David Garlan

March, 1994

CMU-CS-94-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper combines and extends work published as "Beyond Definition/Use: Architectural Interconnection," *Proc. Workshop on Interface Definition Languages*, January 1994, "Using Refinement to Understand Architectural Connection," *Proc. 6th Refinement Workshop*, January 1994, and "Formalizing Architectural Connection," *Proc. International Conference on Software Engineering*, May 1994.

This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the U.S. Government, or Siemens Corporation.

Keywords: software architecture, formal models, model checking, module interconnection, software analysis.

Abstract

As software systems become more complex the overall system structure – or software architecture – becomes a central design problem. An important step towards an engineering discipline of software is a formal basis for describing and analyzing these designs. In this paper we present a theory for one aspect of architectural description: the interactions between components. The key idea is to define architectural connectors as explicit semantic entities. These are specified as a collection of protocols that characterize each of the participant roles in an interaction and how these roles interact. We illustrate how this scheme can be used to define a variety of common architectural connectors. We further provide a formal semantics and show how this leads to a system in which architectural compatibility can be checked in a way analogous to type checking in programming languages.

1 Introduction

As software systems become more complex the overall system structure – or software architecture – becomes a central design problem. Design issues at this level include gross organization and control structure, assignment of functionality to computational units, and high-level interactions between these units. While the design of a good software architecture has always been a significant factor in the success of any large software system, it is only recently that the specific topic of software architecture has been identified as a focus for research and development in workshops [LC93, T+92], government funding, and industrial development [MG92].

The importance of software architecture for practicing software engineers is highlighted by the ubiquitous use of architectural descriptions in system documentation. Most software systems contain a description of the system in terms such as "client-server organization," "layered system," "blackboard architecture," etc. These descriptions are typically expressed informally and accompanied by box and line drawings indicating the global organization of computational entities and the interactions between them.

While these descriptions may provide useful documentation, the current level of informality limits their usefulness. It is generally not clear precisely what is meant by such an architectural description. Hence it may be impossible to analyze the architecture for consistency or infer non-trivial properties about it. Moreover, there is virtually no way to check that a system implementation is faithful to its architectural design.

Evidently, what is needed is a more rigorous basis for describing software architectures. At the very least we should be able to say precisely what is the intended meaning of a box and line description of some system. More ambitiously, we should be able to check that the overall description is consistent in the sense that the parts fit together appropriately. More ambitiously still, we would like a complete theory of architectural description that allows us to reason about the behavior of the system as a whole.

In this paper we describe a first step towards these goals by providing a formal basis for specifying the interactions between architectural components. The essence of our approach is to provide a notation and underlying theory that gives architectural connection explicit semantic status. Specifically, we provide a formal system for specifying architectural *connector types*. The description of these connector types is based on the idea of adapting communications protocols to the description of component interactions in a software architecture.

Of course the use of protocols as a mechanism for describing interactions between parts of a system is not new. However, as we will show, there are three important innovations in our application of this general idea to architectural description. First, we show how the ideas that have traditionally been used to characterize message communication over a network can be applied to description of software interactions. Second, unlike typical applications of protocols we distinguish connector types from connector instances. This allows us to define and analyze architectural connectors independent of their actual use, and then later "instantiate" them to describe a particular system, thereby supporting reuse. Third, we show how a connector specification can be decomposed into parts that simplify its description and analysis. This allows us to localize and automate the reasoning about whether a connector instance is used in a consistent manner in a given system description.

We begin by motivating the need for a theory of architectural connection by examining the limitations of traditional module interconnection languages. We then describe our general model for architectural description and briefly characterize the hard problems in developing a theory of architectural connection to support that model. Next we outline our notation and illustrate through examples how it is used to solve these problems. Having motivated the approach we provide a formal semantics and show how this leads to a system in which architectural compatibility can be checked in a way analogous to type checking in programming languages. Finally, we discuss how our work is related to other approaches to architectural description.

2 The Need for a Theory of Architectural Connection

Large software systems require decompositional mechanisms in order to make them tractable. By breaking a system into pieces it becomes possible to reason about overall properties by understanding the properties of each of the parts. Traditionally, Module Interconnection Languages (MILs) and Interface Definition Languages (IDLs) have played this role by providing notations for describing (a) computational units with well-defined interfaces, and (b) compositional mechanisms for gluing the pieces together.

A key issue in design of a MIL/IDL is the nature of that glue. Currently the predominant form of composition is based on definition/use bindings [PDN86]. In this model each module *defines* or *provides* a set of facilities that are available to other modules, and *uses* or *requires* facilities provided by other modules. The purpose of the glue is to resolve the definition/use relationships by indicating for each use of a facility where its corresponding definition is provided.

This scheme has many benefits. It maps well to current programming languages, since the kinds of facilities that are used or defined can be chosen to be precisely those of an underlying programming language. (Typically these facilities support procedure call and data sharing.) It is good for the compiler, since name resolution is an integral part of producing an executable system. It supports both automated checks (*e.g.*, type checking) and formal reasoning (*e.g.*, in terms of pre- and post-conditions). And, it is in widespread use.

Given all this, one might well ask why anything more is needed. Indeed, the benefits of describing a system in terms of definition/use relationships are so transparent that few people question the basic tenets of the approach.

However, the problem with this traditional approach is that, while it is good for describing *implementation* relationships between parts of a system, it not well-suited to describing the *interaction* relationships that occur in architectural abstractions. The distinction between a description of a system based on "implements" relationships and one based on "interacts" relationships is important for three reasons.

First, the two kinds of relationship have different requirements for abstraction. In the case of implementation relationships it is usually sufficient to adopt the primitives of an underlying programming language – *e.g.*, procedure call and data sharing. In contrast, as noted earlier (and elsewhere [GS93]), interaction relationships at an architectural level of design often involve abstractions not directly provided by programming languages: pipes, event broadcast, client-server protocols, etc. Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system. Hence, the abstractions associated with interactions reflect diverse and potentially complex patterns of communication [Sha93].

Second, they involve different ways of reasoning about the system. In the case of implementation relationships, reasoning typically proceeds hierarchically: the correctness of one module depends on the

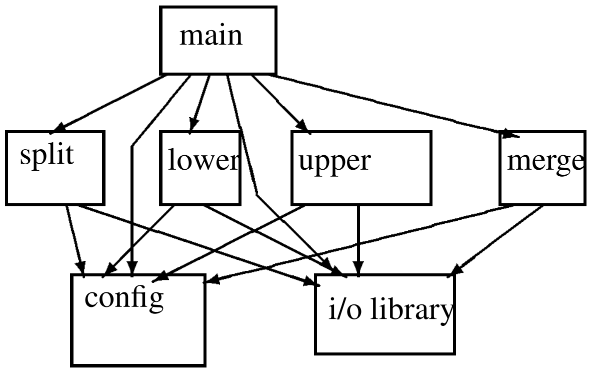

Figure 1: An Implementation Description.

correctness of the modules that it uses. In the case of interaction relationships, the components (or modules) are logically independent of each other: the correctness of each module is independent of the correctness of other modules with which it interacts. Of course, the aggregate system behavior depends on the behavior of its constituent modules and the way that they interact.

Third, they involve different requirements for compatibility checking. In the case of implementation relationships, type checking is used to determine if a use of a facility matches its definition. In the case of interaction relationships, we are more interested in whether protocols of communication are respected. For example, does the reader of a pipe try to read beyond the end-of-input marker; or is the server initialized before a client makes a request of it.

To illustrate these distinctions consider a simple system, *Capitalize*, that transforms a stream of characters by capitalizing alternate characters while reducing the others to lowercase. Let us assume that the system is designed as a pipe-filter system that splits the input stream (using the filter *split*), manipulates each resulting substream separately (using filters *upper* and *lower*) and then remerges the substreams (using *merge*). In a typical implementation of this design we would likely find a decomposition such as the one illustrated in Figure 1. It consists of a set-up routine (*main*), a configuration module (*config*), input/output libraries, as well as modules for accomplishing the desired transformations. The set-up routine depends on all of the other modules, since it must coordinate the transformations and do the necessary hooking up of the streams. Each of the filters uses the configuration module to locate its inputs and outputs, and the i/o library to read and write data.

While useful, this diagram fails to capture the architectural composition of the system. It indicates what modules are present in the system, and to what modules their implementations refer.[1] But it fails to capture the overall system design, or architecture. The pipes in this pipe-filter system are not shown.

An alternative representation of the system is shown in Figure 2. In contrast to the previous description, the interaction relationships (*i.e.*, the pipes) are highlighted while implementation dependencies are suppressed. Of course, for the picture to have any meaning at all there must be a shared understanding of the meaning of the boxes and lines as filters and pipes.

[1]In this simple example we use the term "modules" loosely to represent separable coding units.

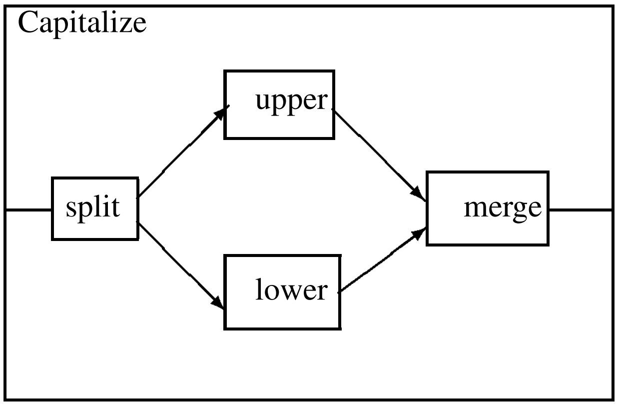

Figure 2: An Architectural Description.

This second description clearly highlights the architectural design and suggests that in order to understand a system it is important to express not only the definition/use dependency relations between implementation "modules," but also to reflect directly the abstract interactions that result in the effective composition of independent components. In particular, to understand and reason about *Capitalize* it is at least as important to know that the output of *upper* is delivered to *merge* as it is to know that it is invoked by *main* and uses *i/o library*.

This example illustrates the three distinctions outlined above. In terms of needs for abstraction, the lines in the first description represent programming language relationships such as "calls" (in the case of the i/o library) or "shares data structure" (in the case of the configuration description). In terms of needs for checking, type checkers will make sure that the required i/o routines exist in the library and that those routines are called with appropriate parameters. For the second description, we are also interested in typing questions: do the interacting filters agree on the type of data passing over the pipe. But, more importantly, we are concerned that the protocols of interaction are respected. For example, we would like to be able to check that a filter either reads or writes to a pipe, but not both; the filters agree on conventions for signaling end of data; the reader does not expect any more data than the writer will provide.

In terms of reasoning about the system, the first description shows a hierarchical decomposition of the system functionality. A specification of the module *main* will be the specification of the behavior of the system as a whole. The first description indicates that the implementation of *main* refers to declarations in the other modules. To demonstrate the correctness of this module, the other modules' specifications must be used, and their correctness shown. In the second description, however, the four filters are represented as logically separate entities that happen to be configured in a particular way. In order for the overall system computation to be correct one would have to consider the composition of the behaviors of the components together with the interactions that are indicated by the lines.

While we have illustrated our points in terms of pipe and filter systems, a similar story could be told for any of the many other architectural idioms and styles that are used to characterize systems: client-server organizations, blackboard systems, interpreters, etc. Unfortunately, the representation of such architectural designs has the current drawback that at present they remain informal depictions. Clearly what is needed is a way to make such diagrams precise in the same way that programming languages (and their associated modularization capabilities) have precise meanings. This involves, at the very least, developing a theory for architectural connection.

3 Requirements for a Theory of Architectural Connection

We take as our starting point a view of architectural description inspired by informal descriptions: a software architecture can be defined as a collection of computational *components* together with a collection of *connectors*, which describe the interactions between the components. While this abstraction ignores some important aspects of architectural description (such as hierarchical decomposition, assignment of computations to processors, and global synchronization and scheduling), it provides a convenient starting point for discussing architectural description.

As a simple example, consider a system organized in a client-server relationship. The components consist of a server and a set of clients. The connectors determine the interactions that can take place between the clients and the servers. In particular, they specify how each client accesses the server. To give a more precise definition of the system we must specify the behavior of the components and show how the connectors define the inter-component interactions.

In this paper we are concerned with providing a formal notation and theory for such architectural connection. Before presenting our solution, however, it is worth highlighting the properties of expressiveness and analytic capability that an appropriate theory and notation should have.

An *expressive* notation for connectors should have three properties. First, it should allow us to specify common cases of architectural interaction, such as procedure call, pipes, event broadcast, and shared variables. Second, it should allow us to describe complex dynamic interactions between components. For example, in describing a client-server connection we might want to say that the server must be initialized by the client before a service request can be made. Third, it should allow us to make fine-grained distinctions between variations of a connector. For instance, consider interactions defined by shared variable access. We would like to be able to distinguish at least the following variants: (a) the shared variable need not be initialized before it is accessed; (b) the shared variable must be initialized by a designated "owner" component before it can be accessed; (c) the shared variable must be initialized by at least one component, but it doesn't matter which.

Descriptive power alone is not sufficient: the underlying theory should also make it possible to *analyze* architectural descriptions. First, we should be able to understand the behavior of a connector independent of the specific context in which it will be used. For example, we should be able to understand the abstract behavior of a pipe without knowing what filters it connects, or even if the components that it connects are in fact filters. Second, we need to be able to reason about compositions of components and connectors. Specifically, our theory should permit us to check that an architectural description is well-formed in the sense that the uses of its connectors are compatible with their definitions. For example, we should be able to detect a mismatch if we attempt to connect a non-initializing client to a server that expects to be initialized. Moreover, we would like these kinds of checks to be automatable. Third, while mismatches should be detectable, we would also like to allow flexibility. For example, (as in Unix) we might want to connect a file to a filter through a pipe, even though the pipe expects a filter on both ends. Hence, independent

```
System SimpleExample
    Component Server
        Port provide [provide protocol]
        Spec [Server specification]
    Component Client
        Port request [request protocol]
        Spec [Client specification]
    Connector C-S-connector
        Role client [client protocol]
        Role server [server protocol]
        Glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

Figure 3: A Simple Client-Server System

specification is analogous to type definitions for programming languages; checking for well-formedness is analogous to the use of type checking to guarantee that all uses of procedures are consistent with their definitions; requirements of flexibility are analogous to subtyping. In the remainder of this paper we show how these goals can be realized.

4 Architectural Description

We begin by describing our general approach to architectural description. In the next section we return to the issue of specifying connectors.

Figure 3 shows how a simple client-server system would be described in the WRIGHT architectural description language.[2] The architecture of a system is described in three parts. The first part of the description defines the *component* and *connector* types. A component type is described (for the purposes of this paper) as a set of *ports* and a *component-spec* that specifies its function. Each port defines a logical point of interaction between the component and its environment.[3] In this simple example Server and Client components both have a single port, but in general a component might might have many ports.

A connector type is defined by a set of *roles* and a *glue* specification. The roles describe the expected local behavior of each of the interacting parties. For example, the client-server connector illustrated above has a client role and a server role. Although not shown in the figure, the client role might describe the client's behavior as a sequence of alternating requests for service and receipts of the results. The server role

[2]The name refers to the architect Frank Lloyd Wright.

[3]Ports are *logical* entities: there is no implication that a port must be realized as a port of a task in an operating system.

might describe the server's behavior as the alternate handling of requests and return of results. The glue specification describes how the activities of the client and server roles are coordinated. It would say that the activities must be sequenced in the order: client requests service, server handles request, server provides result, client gets result.

The second part of the system definition is a set of component and connector *instances*. These specify the actual entities that will appear in the configuration. In the example, there is a single server (s), a single client (c), and a single C-S-connector instance (cs).

In the third part of the system definition, component and connector instances are combined by prescribing which component ports are attached as (or instantiate) which connector roles. In the example, the client request and server provide ports are "attached as" the client and server roles respectively. This means that the connector cs coordinates the behavior of the ports c.request and s.provide. In a larger system, there might be other instances of C-S-connector that define interactions between other ports.

5 Connector Specification

The preceding discussion raises a number of questions. How are ports, roles, and glue defined? What does port instantiation mean? Are there checkable constraints on which ports can be instantiated in which roles? What kinds of analysis can be applied to system configurations? We now provide answers to these questions.

5.1 Process Notation

As outlined above, the roles of a connector describe the possible behaviors of each participant in an interaction, while the glue describes how these behaviors are combined to form a communication. But how do we characterize a "behavior," and how do we describe the range of "behaviors" that can occur?

Our approach is to describe these behaviors as interacting protocols. We use a process algebra to model traces of communication events. Specifically, we use a subset of CSP [Hoa85] to define the protocols of the roles, ports and glue. (In what follows, we will assume that the reader has some familiarity with CSP.)

While CSP has a rich set of concepts for describing communicating entities, we will use only a small subset of these, including:

- **Processes and Events:** A process describes an entity that can engage in communication events.[4] Events may be primitive or they can have associated data (as in e?x and e!x, representing input and output of data, respectively). The simplest process, STOP, is one that engages in no events. The event $\surd$ is used represent the "success" event. The set of events that a process, P, understands is termed the "alphabet of P," or αP.
- **Prefixing:** A process that engages in event e and then becomes process P is denoted $e \rightarrow P$.
- **Alternative:** ("deterministic choice") A process that can behave like P or Q, where the choice is made by the environment, is denoted $P \,\square\, Q$. ("Environment" refers to the other processes that interact with the process.)

[4]It should be clear that by using the term "process" we do not mean that the implementation of the protocol would actually be carried out by a separate operating system process. That is to say, processes are logical entities used to specify the components and connectors of a software architecture.

- **Decision:** ("non-deterministic choice") A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Unlike CSP, however, we restrict the syntax so that only a finite number of process names can be introduced. We do not permit, for example, names of the form $Name_i$, where i can range over the positive numbers.

In process expressions $\rightarrow$ associates to the right and binds tighter than either $\square$ or $\sqcap$. So $e \rightarrow f \rightarrow P \,\square\, g \rightarrow Q$ is equivalent to $(e \rightarrow (f \rightarrow P)) \,\square\, (g \rightarrow Q)$.

In addition to this standard notation from CSP we introduce three notational conventions. First, we use the symbol $\surd$ to represent a successfully terminating process. This is the process that engages in the success event, $\surd$, and then stops. (In CSP, this process is called SKIP.) Formally, $\surd \stackrel{\text{def}}{=} \surd \rightarrow$ STOP.

Second, we allow the introduction of scoped process names, as follows: **let** $Q = expr1$ **in** R.

Third, as in CSP, we allow events and processes to be labeled. The event e labeled with l is denoted $l.e$. The operator ":" allows us to label all of the events in a process, so that $l : P$ is the same process as P, but with each of its events labeled. For our purposes we use the variant of this operator that does not label $\surd$. We use the symbol Σ to represent the set of all unlabeled events.

This subset of CSP defines processes that are essentially finite state. It provides sequencing, alternation, and repetition, together with deterministic and non-deterministic event transitions (but not the cration of an infinite number of process names).

5.2 Connector Description

To describe a connector type we simply provide process descriptions for each of its roles and its glue. As a very simple example, consider the client-server connector introduced earlier.[5] This is how it might be written using the notation just outlined.

connector Service =
  role Client = request!x$\rightarrow$ result?y $\rightarrow$ Client $\sqcap$ $\surd$
  role Server = invoke?x$\rightarrow$ return!y $\rightarrow$ Server $\square$ $\surd$
  glue = Client.request?x$\rightarrow$ Service.invoke!x $\rightarrow$Service.return?y$\rightarrow$Client.result!y$\rightarrow$**glue**
    $\square$ $\surd$

The Server role describes the communication behavior of the server. It is defined as a process that repeatedly accepts an invocation and then returns; or it can terminate with success instead of being invoked. Because we use the alternative operator ($\square$), the choice of invoke or $\surd$ is determined by the environment of that role (which, as we will see, consists of the other roles and the glue).

The Client role describes the communication behavior of the user of the service. Similar to Server, it is a process that can call the service and then receive the result repeatedly, or terminate. However, because we use the decision operator ($\sqcap$) in this case, the choice of whether to call the service or to terminate is

[5]We use simple examples in order to expose the central ideas. The reader should not assume that this indicates an inability to scale to realistic inter-component protocols. For example, see [Jif90] for a representative larger application of CSP to protocol definition.

$$
\begin{array}{l}
\textbf{connector}\ \mathrm{Shared\ Data_1} = \\
\quad \textbf{role}\ \mathrm{User_1} = \mathrm{set}\rightarrow\mathrm{User_1} \sqcap \mathrm{get}\rightarrow\mathrm{User_1} \sqcap \surd \\
\quad \textbf{role}\ \mathrm{User_2} = \mathrm{set}\rightarrow\mathrm{User_2} \sqcap \mathrm{get}\rightarrow\mathrm{User_2} \sqcap \surd \\
\quad \textbf{glue} = \mathrm{User_1.set}\rightarrow\textbf{glue}\ \square\ \mathrm{User_2.set}\rightarrow\textbf{glue} \\
\qquad\qquad \square\ \mathrm{User_1.get}\rightarrow\textbf{glue}\ \square\ \mathrm{User_2.get}\rightarrow\textbf{glue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \mathrm{Shared\ Data_2} = \\
\quad \textbf{role}\ \mathrm{Initializer} = \\
\qquad \textbf{let}\ \mathrm{A} = \mathrm{set}\rightarrow\mathrm{A} \sqcap \mathrm{get}\rightarrow\mathrm{A} \sqcap \surd \\
\qquad \textbf{in}\ \mathrm{set}\rightarrow\mathrm{A} \\
\quad \textbf{role}\ \mathrm{User} = \mathrm{set}\rightarrow\mathrm{User} \sqcap \mathrm{get}\rightarrow\mathrm{User} \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \mathrm{Continue} = \mathrm{Initializer.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{Initializer.get}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User.get}\rightarrow\mathrm{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \mathrm{Initializer.set}\rightarrow\mathrm{Continue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \mathrm{Shared\ Data_3} = \\
\quad \textbf{role}\ \mathrm{Initializer} = \\
\qquad \textbf{let}\ \mathrm{A} = \mathrm{set}\rightarrow\mathrm{A} \sqcap \mathrm{get}\rightarrow\mathrm{A} \sqcap \surd \\
\qquad \textbf{in}\ \mathrm{set}\rightarrow\mathrm{A} \\
\quad \textbf{role}\ \mathrm{User} = \mathrm{set}\rightarrow\mathrm{User} \sqcap \mathrm{get}\rightarrow\mathrm{User} \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \mathrm{Continue} = \mathrm{Initializer.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{Initializer.get}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User.get}\rightarrow\mathrm{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \mathrm{Initializer.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\quad \square\ \mathrm{User.set}\rightarrow\mathrm{Continue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \mathrm{Bogus} = \\
\quad \textbf{role}\ \mathrm{User_1} = \mathrm{set}\rightarrow\mathrm{User_1} \sqcap \mathrm{get}\rightarrow\mathrm{User_1} \sqcap \surd \\
\quad \textbf{role}\ \mathrm{User_2} = \mathrm{set}\rightarrow\mathrm{User_2} \sqcap \mathrm{get}\rightarrow\mathrm{User_2} \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \mathrm{Continue} = \mathrm{User_1.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User_2.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User_1.get}\rightarrow\mathrm{Continue} \\
\qquad\qquad\qquad\qquad\qquad \square\ \mathrm{User_2.get}\rightarrow\mathrm{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \mathrm{User_1.set}\rightarrow\mathrm{Continue} \\
\qquad\qquad\quad \square\ \mathrm{User_2.set}\rightarrow\mathrm{Continue}\ \square\ \surd
\end{array}
$$

Figure 4: Several Shared Data Connectors

determined by the role process itself. Comparing the two roles, note that the two choice operators allow us to distinguish formally between situations in which a given role is *obliged* to provide some services – the case of Server – and the situation where it may take advantage of some services if it chooses to do so – the case of Client.

The **glue** process coordinates the behavior of the two roles by indicating how the events of the roles work together. Here **glue** allows the Client role to decide whether to call or terminate and then sequences the remaining three events and their data.

The example above illustrates that the connector description language is capable of expressing the traditional notion of providing and using a set of services – the kind of connection supported by import/export clauses of module interconnection. To take a more interesting example – one in which the power of the approach becomes evident – consider the problem of specifying a "shared variable" connector in such a way that requirements of initialization are made explicit.

Figure 4 illustrates four possible specifications.[6] As we see from the glue specifications, the four specifications are quite similar. Each role has two events, get and set, indicating a read or write of the shared value, respectively. After a possible initialization, all of the connectors are the same. Any of the roles can either get or set the value repeatedly, quitting at any time. The overall communication is complete only when all participants are done with the data. Each differs, however, in how initialization is handled.

The first specification, Shared Data$_1$, indicates that the data does not require an explicit initialization value. Shared Data$_1$ permits either participant to get or set as the first event. The second specification,

[6] In this set of examples, for simplicity we ignore the data behavior of the connector. In a fuller shared data connector description, each event would have a data parameter.

```
connector Pipe =
    role Writer = write→Writer ⊓ close→ √
    role Reader = let ExitOnly = close→ √
                  in let DoRead = (read→Reader □ read-eof→ExitOnly)
                  in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read→ReadOnly
                        □ Reader.read-eof →Reader.close → √
                        □ Reader.close→ √
        in let WriteOnly = Writer.write→WriteOnly □ Writer.close→ √
        in Writer.write→glue □ Reader.read→glue
         □ Writer.close→ReadOnly □ Reader.close→WriteOnly
```

Figure 5: A Pipe Connector

Shared Data_2, indicates that there is a distinguished role Initializer that must supply the initial value. The Initializer agrees to set the value before getting it, and the glue ensures that this will occur before the other participant (User) accesses the variable, for either a get or a set. The third alternative, Shared Data_3, is similar to the second in that it has an explicit Initializer role that promises to set initially, but it does not require that the other participant wait for that initialization to proceed. If the User begins with a set, then it can continue without waiting for Initializer. The final alternative, Bogus, seems reasonable – the connector glue requires that one of the participants initialize the variable, but does not specify which one. If either begins with a set, then that event will occur first and the communication can continue without any problem. If, however, each participant attempts, legally, to perform an initial get, then the connector will deadlock. We will return to the important problem of detecting such anomalous behavior in Section 8.2.

As a more complex example, consider the pipe connector type. It might appear to be a simple matter to define a pipe: both the writer and the reader decide when and how many times they will write or read, after which they will each close their side of the pipe. In fact, the writer role is just that simple. The reader, on the other hand, must take other considerations into account. There must be a way to inform the reader that there will be no more data. A pipe connector that describes this behavior is shown in Figure 5.

6 Connector Semantics

Informally, the meaning of a connector description is that the roles are treated as independent processes, constrained only by the glue, which serves to coordinate and interleave the events.

To make this idea precise we use the CSP parallel composition operator, $\|$, for interacting processes. The process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2. That is, for the events in the intersection of the processes' alphabets, both processes must agree to engage in the event. We can then take the meaning of a connector description to be the parallel interaction of the glue and the roles, where the alphabets of the roles and glue are arranged so that the desired coordination occurs.

Definition 1 The *meaning of a connector description* with roles $R_1, R_2, \ldots, R_n$, and glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1\mathbf{:}R_1 \parallel \mathrm{R}_2\mathbf{:}R_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}R_n)$$

where R_i is the (distinct) name of role R_i, and

$$\alpha Glue = \mathrm{R}_1\mathbf{:}\Sigma \cup \mathrm{R}_2\mathbf{:}\Sigma \cup \ldots \cup \mathrm{R}_n\mathbf{:}\Sigma \cup \{\surd\}.$$

In this definition we arrange for the glue's alphabet to be the union of all possible events labeled by the respective role names (*e.g.* Client, Server), together with the $\surd$ event. This allows the glue to interact with each role. In contrast, (except for $\surd$) the role alphabets are disjoint and so each role can only interact with the glue. Because $\surd$ is not relabeled, all of the roles and glue can (and must) agree on $\surd$ for it to occur. In this way we ensure that successful termination of a connector becomes the joint responsibility of all the parties involved.

7 Ports and Connector Instantiation

Thus far we have concerned ourselves with the definition of connector types. To complete the picture we must also describe the ports of components and how those ports are attached as specific connector roles in the complete software architecture. (See Figure 3.)

In WRIGHT, component ports are also specified by processes: The port process defines the expected behavior of the component at that particular point of interaction. For example, a component that uses a shared data item only for reading might be partially specified as follows:

component DataUser =
    port DataRead = get$\rightarrow$DataRead $\sqcap$ $\surd$
    other ports...

Since the port protocols define the actual behavior of the components when those ports are associated with the roles, the port protocol takes the place of the role protocol in the actual system. Thus, an attached connector is defined by the protocol that results from the replacement of the role processes with the associated port processes. More formally,

Definition 2 The meaning of attaching ports $P_1 \ldots P_n$ as roles $R_1 \ldots R_n$ of a connector with glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1\mathbf{:}P_1 \parallel \mathrm{R}_2\mathbf{:}P_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}P_n).$$

Note that implicit in this definition of attachment is the idea that port protocols need not be identical to the role protocols that they replace. This is a reasonable decision because it allows greater opportunities for reuse. In the above example, the DataUser component should be able to interact with another component (via a shared data connector) even though it never needs to set. As another example, we would expect to be able to attach a File port as the Reader role of a pipe (as is commonly done in Unix when directing the output of a pipe to a file).

But this raises an important question: when is a port "compatible" with a role? For example, it would be reasonable to forbid DataRead to be used as the Initializer role for Shared Data_2 and Shared Data_3 connectors, since these require an initial set; clearly DataRead will never provide this event. We consider this issue in the next section.

8 Analyzing Architectural Descriptions

We now consider the kinds of analysis and checking that are made possible by our connector notation and formalism.

8.1 Compatibility (of a port with a role)

An important reason to provide specific definitions of role protocols is to answer the question "what ports may be used in this role?" At first glance it might seem that the answer is obvious: simply check that the port and role protocols are equivalent. But as illustrated earlier, it is important to be able to attach a port that is not identical to the role. On the other hand, we would like to make sure that the port fulfills its obligations to the interaction. For example, if a role requires an initialization as the first operation (*cf.*, Figure 4), we would like to guarantee that any port actually performs it.

Informally, we would like to be able to guarantee that an attached port process always acts in a way that the corresponding role process is capable of acting. This can be recast as follows: When in a situation predicted by the protocol, the port must always continue the protocol in a way that the role could have.

In CSP this intuitive notion is captured by the concept of refinement. Roughly, process P_2 refines P_1 (written $P_1 \sqsubseteq P_2$) if the behaviors of P_1 include those of P_2. Technically, the definition is given in terms of the failures/divergences model of CSP [Hoa85, Chapter 3].

However, it is not possible to use CSP's definition of refinement directly to define port-role compatibility for two reasons. The first is the technicality that CSP's $\sqsubseteq$ relation assumes that the alphabets of the compared processes are the same. We can handle this problem simply by augmenting the alphabets of the port and role processes so that they are identical. This is easily accomplished using the CSP operator for extending alphabets of processes: P_{+B} extends the alphabet of process P by the set B.[7]

The second reason is that even if the port and role have the same alphabet it may be that the port process is defined so that incompatible behavior is possible in general, but would never arise in the context of the connector to which it is attached. For example, suppose a component port has the property that it must be initialized before use, but that it will crash if it is initialized twice. If we put this in the context of a connector that *guarantees* that at most one initialization will occur (*e.g.*, see Figure 4), then the anomalous situation will not arise. Although we would not condone such a component definition, we would expect that formally the port is compatible with the role.

Thus to evaluate compatibility we need to concern ourselves only with the behavior of the port *restricted to the contexts in which it might find itself.* Technically we can achieve this result by considering the new process formed by placing the port process in parallel with the deterministic process obtained from the role. For a role R, we denote this latter process $det(R)$. Formally,

Definition 3 $det(R) = (\alpha R, \{(t, s) \mid t \in traces(R) \wedge \forall\, e \in s \bullet t ^\frown \langle e \rangle \notin traces(R)\}, \varnothing)$

Thus (using "\" as set difference) we are led to the following definition of compatibility:

Definition 4 P **compat** R ("P is compatible with R") if $R_{+(\alpha P \setminus \alpha R)} \sqsubseteq P_{+(\alpha R \setminus \alpha P)} \parallel det(R)$.

[7] Formally, $P_{+B} = (P \| STOP_B)$.

Using these definitions, we see that port $DataRead = \text{get} \rightarrow DataRead \sqcap \surd$ is compatible with role $User = \text{set} \rightarrow User \sqcap \text{get} \rightarrow User \sqcap \surd$ because the role could always decide to engage in get. On the other hand, *DataRead* is *not* compatible with role $Initializer = \textbf{let } Continue = \ldots \textbf{ in } \text{set} \rightarrow Continue$ because this latter role indicates that at least initially, set must be offered to the connector.

8.2 Deadlock Freedom

While intuitively motivated, our definition of compatibility might at first glance appear obscure, and the skeptical reader may well ask "What good is it anyway?" Like type correctness for programming languages, compatibility for architectural description is intended to provide certain guarantees that the system is well formed. By that standard, the proof of utility for compatibility must be that it does, in fact, guarantee that important properties hold in a "compatible" system and that, moreover, it is possible to provide practical tools for compatibility checking. In the remainder of this section and the next section we demonstrate these results.

An important property of any system of interacting parts is that it is free from deadlock. Informally, in terms of connectors this means that two components do not get "stuck" in the middle of an interaction, each port expecting the other to take some action that can never happen. On the other hand, we *do* want to allow terminating behaviors in which all of the ports (and the glue) agree on success. For example, in a client-server connection it should be possible for a client to terminate the interaction, provided it does so at a point expected by the server.

We can make this precise by saying that a connector process C is free from deadlock if whenever it is in a situation where it cannot make progress (formally, its refusal set is its entire alphabet), then the last event to have taken place must have been the success event. Thus we have:

Definition 5 A connector C is *deadlock-free* if for all $(t, ref) \in failures(C)$ such that $ref = \alpha C$, then $last(t) = \surd$.

Such a property is only useful if it is preserved across connector instantiation. That is, we would like to be able to claim that a deadlock-free connector remains deadlock-free when instantiated with compatible ports.

By definition of port instantiation, a deadlock-free connector will remain deadlock-free when its roles are instantiated with ports that exactly match the roles. But as we have argued above, ports and roles need not be identical. Less obvious, but equally true, is the fact that if ports are strict refinements of the roles then deadlock freedom is also preserved. This follows from the monotonic nature of process refinement, which requires the failures of a refinement to be a subset of the failures of the process it is refining. In other words, the refined process can't refuse to participate in an interaction if the role could not also have refused.

But we have deliberately chosen a weaker notion of refinement in order to provide greater opportunities for reuse of the connector. Because the port need only be considered a refinement when restricted to the traces of the role, it is possible that it may allow potentially deadlocking behavior, even though this behavior would never occur in the context of the role that it is playing.

Consequently, it is not immediately clear whether deadlock-freedom is preserved across compatible port substitutions. In fact, it is not. The problem arises if the glue permits behaviors outside the range of those defined by the roles of the connector. Suppose, for example, that the glue allows a behavior of the form "$\text{R1}.good \rightarrow \surd \,\square\, \text{R1}.crash \rightarrow \text{STOP}$" and that the event *crash* is not permitted by role $R1$. $R1$'s behavior would look like "*good* $\rightarrow \surd$." Then the connector could be deadlock-free (in the sense defined above) because the event *crash* will never be permitted. Now consider a port that contains the *crash* behavior (*i.e.*, $good \rightarrow \surd \sqcap crash \rightarrow$ STOP). This port is compatible with role $R1$ because it won't refuse *good*. But the connector can deadlock if the port is substituted for role $R1$, because neither the glue nor the port will refuse *crash*.

To avoid this possibility we need to impose further restrictions on the glue. Specifically, we define a *conservative* connector to be one for which the glue traces are a subset of the possible interleavings of role traces.

Definition 6 A connector $C = Glue \parallel (\mathrm{R}_1\mathbf{:}r_1 \parallel \mathrm{R}_2\mathbf{:}r_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}r_n)$ is *conservative* if $traces(Glue) \subseteq traces(\mathrm{R}_1\mathbf{:}r_1 \| \mathrm{R}_2\mathbf{:}r_2 \| \ldots \| \mathrm{R}_n\mathbf{:}r_n)$.

This means that the glue is responsible for preventing behaviors that are not covered by the role specifications.

We can now state the theorem that ties all of these ideas together. It states that compatibility ensures the deadlock freedom of any instantiated well-formed connector (*i.e.*, one that is deadlock free and conservative).

Theorem 1 *If a connector* $C = Glue \parallel (\mathrm{R}_1\mathbf{:}R_1 \parallel \mathrm{R}_2\mathbf{:}R_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}R_n)$ *is conservative and deadlock-free, and if for* $i \in \{1..n\}$, P_i **compat** R_i, *then* $C' = Glue \parallel (\mathrm{R}_1\mathbf{:}P_1 \parallel \mathrm{R}_2\mathbf{:}P_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}P_n)$ *is deadlock-free.*

The significance of this theorem is twofold. First, it tells us that local compatibility checking is sufficient to maintain deadlock freedom for any instantiation. Second, it provides a kind of soundness check for our definition of compatibility: under any execution, a compatibly instantiated architectural description retains certain properties.

Proof of Theorem 1

A lemma makes the proof simpler:

Lemma 1 *if* $traces(P) \upharpoonright \alpha Q \subseteq traces(Q)$, *then* $P \| det(Q) = P$

This lemma is essentially an observation that a deterministic process ($det(Q)$) serves only to restrict the traces of another process when composed using $\|$. If the traces are already so restricted, the composition has no effect.

We now prove the theorem by contradiction:
If the instantiated connector, C', is not deadlock-free, then by our definition of deadlock there must be some trace t of C' such that

$$(t, \alpha C') \in failures(C') \text{ and } last(t) \neq \surd$$

$$\alpha(\mathrm{R}_1\mathbf{:}P_1 \| \ldots \| \mathrm{R}_n\mathbf{:}P_n) \subseteq \alpha Glue \qquad \text{[definition of } \alpha Glue]$$

(1) $\Rightarrow t \in traces(Glue)$

(2) $traces(Glue) \subseteq traces(\mathrm{R}_1\mathbf{:}R_1 \| \ldots \| \mathrm{R}_n\mathbf{:}R_n)$ [C conservative]

(1) $\wedge$ (2)

$$\Rightarrow t \in traces(\mathrm{R}_1\mathbf{:}R_1 \| \ldots \| \mathrm{R}_n\mathbf{:}R_n)$$

$$\Rightarrow t \in traces(C) \qquad [\text{semantics of } \|]$$

$$(3) \qquad \Rightarrow (t, \alpha C) \notin failures(C) \qquad [C \text{ is deadlock-free}, last(t) = \surd]$$

$$R_i \sqsubseteq P_i \| det(R_i) \qquad [P_i \textbf{ compat } R_i]$$

$$C = Glue \| \mathrm{R}_1\mathbf{:}R_1 \| \ldots \| \mathrm{R}_n\mathbf{:}R_n \qquad [\text{definition of C}]$$

$$\sqsubseteq Glue \| \mathrm{R}_1\mathbf{:}(P_1 \| det(R_1)) \| \ldots \| \mathrm{R}_n\mathbf{:}(P_n \| det(R_n)) \qquad [\|, L : \text{monotonic}]$$

$$= Glue \| \mathrm{R}_1\mathbf{:}det(R_1) \| \ldots \| \mathrm{R}_n\mathbf{:}det(R_n) \| \mathrm{R}_1\mathbf{:}P_1 \| \ldots \| \mathrm{R}_n\mathbf{:}P_n$$

$$[\| \text{ symmetric}, L : \text{distributes over } \|]$$

$$= Glue \| \mathrm{R}_1\mathbf{:}P_1 \| \ldots \| \mathrm{R}_n\mathbf{:}P_n \qquad [C \text{ conservative, lemma 1)}]$$

$$= C' \qquad [\text{definition of } C']$$

$$C \sqsubseteq C'$$

$$\Rightarrow failures(C') \subseteq failures(C) \qquad [\text{definitions of } \sqsubseteq]$$

$$(4) \qquad \Rightarrow (t, \alpha C) \in failures(C) \qquad [(t, \alpha C) \in failures(C')]$$

But (4) contradicts (3). C' must therefore be deadlock-free.

9 Automating Compatibility Checking

As we have indicated, an important motivation for this work is the potential for automating compatibility checks. To achieve this, we have constrained our use of the CSP notation in two ways. First, we have restricted the notation such that our processes will always be finite. (Of course, infinite traces are still possible even though we can't create an infinite number of processes.) This means that we can use Model Checking technology [B⁺90] to verify properties of the processes and to check relationships between processes. Second, we have expressed our checks as refinement tests on simple functions of the described processes. In other words, we can express our tests as checks of the predicate $P \sqsubseteq Q$ for appropriately constructed finite processes P and Q. This permits us to apply the emerging technology of automated verification tools to make these checks.

To illustrate, we show how we would check the compatibility of *DataRead* with the role *User* using FDR [For92], a commercial tool designed to check refinement conditions for finite CSP processes. First, to use FDR we must translate our notation to fit the variant of CSP used in this tool. Recall that in our notation the processes are:

$$DataRead = \text{get} \rightarrow DataRead \sqcap \surd$$
$$User = \text{set} \rightarrow User \sqcap \text{get} \rightarrow User \sqcap \surd$$

These are encoded in FDR[8] as:

[8] We use the event name `seta` instead of `set` to avoid a name clash with a reserved keyword of FDR.

```
DATAREAD = (get -> DATAREAD) |~| TICK
USER = (seta -> USER) |~| (get -> USER) |~| TICK
```

To test the compatibility of *DataRead* with *User*, we must determine whether

$$User_{+(\alpha DataRead \setminus \alpha User)} \sqsubseteq DataRead_{+(\alpha User \setminus \alpha DataRead)} \parallel det(User)$$

Because $\alpha DataRead \subseteq \alpha User$, it follows that $User_{+(\alpha DataRead \setminus \alpha User)}$ is trivial:

```
USERplus = USER
```

To encode $DataRead_{+(\alpha User \setminus \alpha DataRead)}$, we must encode the interaction with $STOP_{\{\mathrm{set}\}}$:

```
DATAREADplus = DATAREAD [|{seta}|]STOP
```

Next we encode *det*(*User*). To do this, we change the nondeterministic $\sqcap$ to the deterministic $\square$:

```
detUSER = (seta -> detUSER) [] (get -> detUSER) [] TICK
```

This leaves only the encoding of the interaction $DataRead_{+(\alpha User \setminus \alpha DataRead)} \parallel det(User)$

```
DATAREADpD = DATAREADplus [|{seta,get,tick}|] detUSER
```

These processes can then be checked for compatibility by giving FDR the command:

```
Check "USERplus" "DATAREADpD"
```

As with compatibility checking, conservatism and deadlock-freedom can be checked by tools such as FDR. The test for conservatism is a straightforward use of trace refinement, for which FDR provides `CheckTrace`. Similarly, deadlock-freedom can be expressed as a refinement check of the most nondeterministic deadlock-free process.

Applying these checks to the examples in this paper, we easily confirmed that only the connector *Bogus* (Figure 4) can deadlock. However, we also unexpectedly discovered that both $Shared\,Data_2$ and $Shared\,Data_3$ are not conservative. Until we ran the checks we had failed to notice that the glue of these connectors permits an immediate $\surd$, whereas the role *Initializer* prevents this. This dramatically illustrated for us the benefits of automated checking, even for such relatively simple examples.

10 Extending the Glue with Trace Specifications

As we have shown, by restricting protocol specifications to be finite state, we permit automated compatibility checking. (This is analogous to the use of type declarations in conventional module interface languages: type declarations typically provide only partial semantic information, but support automated checks.) But often a finite state protocol can only approximate a particular interaction. For example, in Figure 5, although the glue expresses the importance of the close and read-eof events, it does not capture the first-in/first-out data semantics of the pipe protocol.

To make it possible to express these richer notions, we permit the glue to be augmented with a trace specification. For example, the pipe could be described as in Figure 6. The **spec** indicates that every read

```
connector Pipe =
    role Writer = write!x→Writer ⊓ close→ √
    role Reader = let ExitOnly = close→ √
                  in let DoRead = (read?x→Reader [] read-eof→ExitOnly)
                  in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read!y→ReadOnly
                        [] Reader.read-eof→Reader.close→ √
                        [] Reader.close→ √
        in let WriteOnly = Writer.write?x→WriteOnly [] Writer.close→ √
        in Writer.write?x→glue [] Reader.read!y→glue
          [] Writer.close→ReadOnly [] Reader.close→WriteOnly
    spec ∀ Reader.read_i!y • ∃ Writer.write_j?x • i = j ∧ x = y
        ∧ Reader.read-eof ⇒ (Writer.close ∧ # Reader.read = # Writer.write)
```

Figure 6: A Pipe Connector Augmented with a Trace Specification

event by the Reader corresponds to a matching write event by the Writer. This specification also indicates that the read-eof event will occur after the Reader has read all of the data that the Writer will supply.

To make precise the use of such trace specifications we must say how they affect the meaning of the connector. The obvious intuition is that such specifications should constrain the original protocol to the traces that satisfy the given predicate. To achieve this, we define the overall *Glue* process to be the finite **glue** process restricted to the traces permitted by the trace specification. Formally, we achieve this by defining an interaction with the deterministic process obeying the specification.

Definition 7 For any trace predicate, P, $proc(P)_A$ is the deterministic process such that $\alpha proc(P)_A = A$ and $traces(proc(P)_A) = \{t : A* \mid P\}$

Definition 8 for a connector C with glue protocol **glue** and trace specification S,
$Glue = \mathbf{glue} \| proc(P)_{\alpha Glue}$, where $\alpha Glue$ is as in definition 1.

Of course, it is no longer possible to provide automatable checks about the resulting *Glue*. Instead the specifier has an obligation to show that the augmented protocol is deadlock free and conservative.

11 Comparisons to Other Approaches

Module Interface Languages

As we argued earlier, module interface languages, with their roots in programming languages, deal primarily with definition/use relationships. As such, the issues that they address are largely orthogonal to issues that WRIGHT addresses. Indeed, to describe an *implementation* of one of the components in a WRIGHT description, one would likely use a modern module interconnection language to define the code units and their dependencies.

On the other hand, because until recently software developers have had *only* traditional module notations to describe their systems, they have used the import/export facilities of these languages to define architectural structure as well. The result is that such "architectural" descriptions are necessarily limited to the interaction primitives of the programming language: usually procedure call and data sharing. In this respect, our work provides a significant improvement in expressiveness and analytic capability. It improves expressiveness because it allows the designer to specify new kinds of interactions that are not bound by the limitations of the programming model. Moreover, the description of these interactions includes specification of dynamic behavior (*i.e.*, protocols). The inclusion of dynamic properties allows more properties to be checked than, say, signature matching alone.

Recent research on module interconnection has introduced a number of new mechanisms and richer notions of module interconnection [Per87, Pur94, Rei90]. These primarily serve to extend the basic vocabulary of connection, rather than to give ways to define new kinds of connection (as does our work). Some (such as [Per87]) are also concerned with increasing the semantic content and the checkability of interfaces, and to that extent they share some of our general goals.

Recent work by Lam and Shankar [LS94] has used protocols to provide higher level guarantees about the correct composition of modules. Like our work, Lam and Shankar extend the interface specifications to the ordering of events and distinguish between those aspects of an interface that represent *assumptions* of a module and those that are *promises* by the module. They also use this distinction to provide a definition of compatibility between modules that is more permissive than exact matching. Their work differs from ours in that, by focusing on composition for MILs (rather than software architecture), their model does not have explicitly defined connectors. They also assume that interfaces either represent service providers or service users. This asymmetry reflects the definition/use orientation of their work. Finally, while they use the distinction between input and output events to capture the assumption/promise dichotomy, we use the distinction between deterministic and nondeterministic choice.

Description of Software Architecture

A number of other architectural representation languages have been proposed. Rapide [L+92] uses Posets as a formal basis for architectural description, and supports certain static interface checks, as well as dynamic checks for satisfaction of predicates over system traces. In Rapide connectors are modeled by defining new kinds of components, whereas in WRIGHT connectors function as composition operators.

The UNICON language [S+94] is similar in spirit to WRIGHT. In particular, UNICON shares the notion that connectors are first class entities defined, in part, by a set of roles. However, while UNICON provides placeholders for formal specification of connectors, it does not itself support any particular formal notation. In that sense WRIGHT and UNICON are complementary: our connector specifications could be combined with the overall architectural descriptive notation of UNICON.

Other architectural description languages have been proposed for specialized domains [MG92, DAR90]. These languages increase their analytic and expressive leverage by specializing to a particular family of systems, thereby trading generality for power.

Finally, over the past decade numerous software design languages have been proposed — many of them graphical. These are concerned with the gross decomposition of systems (for example, [Cam89]). As with domain-specific approaches to software architecture, such notations often provide strong support for certain classes of systems. But they do not provide general mechanisms for formal description or analysis of architectural connection.

Other Formal Models

Other models of concurrency could have been used to define the semantics of connectors, including state machines, pre- and post-conditions, and Petri nets. We investigated several state machine approaches such as I/O Automata [LT88], StateCharts [Har87], SMV [C+86], and SDL [Hol91]. While these systems have been used to model protocols and have well-defined mechanisms for composition, we favored the use of CSP for three reasons. First, it has a semantic basis (in terms of traces, divergences, and refusals) that makes it ideal for characterizing problems of connector deadlock and for expressing port-role compatibility as a kind of refinement. Indeed, CSP is the only formal notation for concurrent systems that has both deterministic and non-deterministic choice operators. Second, it provides a powerful calculus for composing systems in terms of parallel composition. Finally, it has industrial-strength tools (such as FDR) for automated analysis.

The choice of CSP, however, does limit us to describing a certain class of properties. For example, we cannot handle properties such as timing behavior of interactions because CSP's semantic model is not rich enough. To address such properties, one can imagine retaining the general descriptive framework (of ports, roles, glue, and glue specification) for connectors, but replacing CSP with an alternative formalism.

Refinement of Protocols

Traditionally, research on protocols has been concerned with developing algorithms to achieve certain communication properities – such as reliable communication over a faulty link. Having developed such an algorithm, the protocol designer assumes that the participants will precisely follow the algorithm specified by the protocol.

Our use of protocols differs in two significant ways. First, our connector protocols specify a set of obligations, rather than a specific algorithm that must be followed by the participants. This allows us to admit situations in which the actual users of the protocol (*i.e.*, the ports) can have quite different behavior than that specified by the connector class (via its roles). This approach allows us to adopt a building-block approach, in which connectors are reused, the context of reuse determining the actual behavior that occurs.

The second major difference is that our approach provides a specific way of structuring the description of connector protocols – namely, separation into roles and glue. The benefit of adopting this structured approach is that it allows us to localize the checking of compatibility when we use a connector in a particular context.

There has been some work that, like ours, exploits the fact that in a constrained situation, the criteria of refinement can be weakened without compromising substitutability (*e.g.*, [Jac89]). However, that work uses refinement to indicate *substitution* of one process for another, in contrast to our work, in which the role serves as a *specification* for the properties of the port (and hence it *defines* the context in which it may be used as well as the properties that the port must have).

Another related use of protocols is in work on extending object-oriented systems with protocols over an object's methods. Nierstrasz [Nie93] extends object class definitions to include a finite-state process over the methods of the object, and defines a subtyping relation and instantiation rules that are similar to our ideas of compatibility. While the motivation is similar to ours, Nierstrasz considers only one kind of component interaction: method invocation. Moreover, the refinement relations that define subtyping and instantiation differ from our tests in that they are specific to a single class of interaction.

Yellin [Yel94] also defines a model of object composition based on protocols. He provides a more restrictive definition of object compatibility and then increases the flexibility of the model using *software adaptors*. These adaptors have some of the flavor of our connectors, but are less general, since purpose is to increase the flexibility of object composition – not to provide a separable and explicit definition of an interaction.

12 Conclusions

A significant challenge for software engineering research is to develop a discipline of software architecture. This paper takes a step towards that goal by providing a formal basis for describing and reasoning about architectural connection. The novel contributions of our approach are:

- The treatment of connectors as types that have separable semantic definitions (independent of component interfaces), together with the notion of connector instantiation.
- The partitioning of connector descriptions into roles (which define the behavior of participants) and glue (which coordinates and constrains the interactions between roles).
- The separation of the semantic definition into two parts: protocols that are sufficiently constrained to be automatically checked for consistency and compatibility, and auxiliary specifications that can capture other properties of a connector.
- The development of formal machinery for automatable compatibility checking of architectural descriptions, thereby making many of the benefits of module interface checking available to designers of software architectures.

In developing this basis for connectors we have adapted the more general theory of process algebras and shown how it can be specialized to the problem of connector specification. While this approach limits the generality of that theory, we argue that it makes the techniques both accessible and practical. It is accessible because semantic descriptions are syntactically constrained to match the problem. It is practical because by limiting the power of expression, we permit automated checking.

There remain many problems of architectural design that this paper does not directly address. In particular, our work on connectors does not explicitly deal with issues associated with global architectural constraints such as global synchronization, scheduling, or global analysis of deadlock. However, the work does open the door to a number of direct extensions that will broaden its applicability even further. These include: operators for building complex connectors out of simpler ones and a theory of connector refinement.

Acknowledgements

We would like to thank Gregory Abowd, Stephen Brookes, Tony Hoare, Daniel Jackson, Eliot Moss, John Ockerbloom, Mary Shaw, and Jeannette Wing for their comments on earlier versions of this paper.

References

[B+90] J. Burch et al. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Symposium on Logic in Computer Science*, June 1990.

[C+86] E. Clarke et al. Automatic verification of finite state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2), April 1986.

[Cam89] John Cameron. *JSP and JSD: the Jackson Approach to Software Development*. IEEE Computer Society Press, 1989.

[DAR90] *Proc. Workshop on Domain-Specific Software Architectures*. CMU Software Engineering Institute, 1990.

[For92] *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems (Europe) Ltd., Oxford, England, 1.2β edition, October 1992.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, I, 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, (8), 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Jac89] Jeremy Jacob. Refinement of shared systems. In John A. McDermid, editor, *The Theory and Practice of Refinement*. 1989.

[Jif90] He Jifeng. Specification and design of the X.25 protocol: A case study in csp. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*. Addison-Wesley, 1990.

[L+92] David C. Luckham et al. Partial orderings of event sets and their application to prototyping concurrent timed systems. 1992.

[LC93] David Lamb and Sandra Crocker, editors. *Proc. of the Workshop on Studies of Software Design*. Queens Univ. Dept. of Comp. and Inf. Sci., 1993. no. ISSN-0836-0227-93-352.

[LS94] Simon S. Lam and A. Uday Shankar. A theory of interfaces and modules i—compostion theorem. *IEEE Transactions on Software Engineering*, 20(1), January 1994.

[LT88] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, 1988.

[MG92] Erik Mettala and Marc H. Graham, editors. *The Domain-Specific Software Architecture Program*. Number CMU/SEI-92-SR-9. CMU Software Engineering Institute, June 1992.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA '93*, volume 28, 1993.

[PDN86] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[Per87] Dewayne E. Perry. Software interconnection models. In *Proc. 9th ICSE*, 1987.

[Pur94] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[Rei90] S.P. Reiss. Connecting tools using message passing in the Field Environment. *IEEE Software*, 7(4), July 1990.

[S+94] Mary Shaw et al. Abstractions for software architecture and tools that support them. Submitted for publication, February 1994.

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

[T+92] W.F. Tichy et al., editors. *Proc. Dagstuhl Worshop on Future Directions in Software Engineering*, 1992.

[Yel94] Daniel M. Yellin. Interfaces, protocols, and the semi-automatic construction of software adaptors. Technical Report RC19460, IBM T.J. Watson Research Center, 1994.