A Formal Basis for Architectural Connection

Robert Allen David Garlan

January 1997

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

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 formal approach to one aspect of architectural design: 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

For complex software systems the overall system structure—or software architecture—emerges as a critical 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 [SG96].

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 architectural descriptions. Hence it may be impossible to analyze an architecture for consistency or determine non-trivial properties of it. Moreover, there is 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 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 describe a formal system for specifying architectural *connector types*. The description of these connector types is based on the idea of characterizing the protocols of interaction between architectural components. The formal underpinnings for protocol definition are provided by CSP [Hoa85].

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. This allows us to apply existing theory and analytical techniques to a new practical domain. 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 solves 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 the key design decisions on which this work is based, compare the work to other approaches to architectural description, and outline future directions for research in this area.

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 are a central concern of architectural descriptions. Specifically, 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, 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 determines how that computation is combined with others in the overall system. Thus the abstractions associated with interactions reflect the diverse and potentially complex patterns of communication that characterize architectural descriptions [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 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 component is independent of the correctness of other components with which it interacts. Of course, the aggregate system behavior depends on the behavior of its constituent components and the way that they interact.

Third, they involve different requirements for checking consistency and completeness of a system description. In the case of implementation relationships, type checking is used to determine if a use of a facility is consistent with its definition. Completeness is typically checked by a loader, which guarantees that every facility required by one module is provided by some other module. In the case of interaction relationships, we are more interested in whether protocols of communication are consistent. 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. Similarly, with respect to interaction relationships, the completeness of a system is concerned with whether the assumptions of each component about the rest of the system have been met, and whether all participants are present in the specified interactions.

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 how the implementations of one module depend on others.[1] But it fails to capture the overall system design, or architecture. In particular, the pipes in this pipe-filter system are not shown directly.

An alternative representation of the system is illustrated 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.

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

[1] Here we use the term "modules" loosely to represent separable coding units.

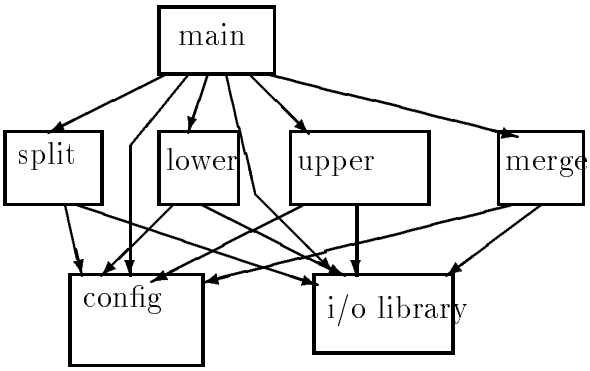

Figure 1: An Implementation Description.

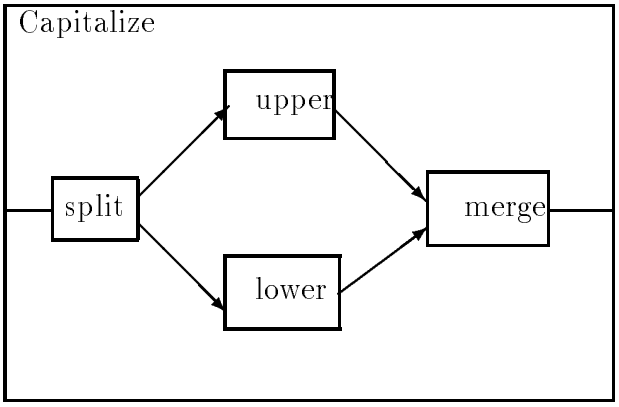

Figure 2: An Architectural Description.

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 upper 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 (architectural) 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 provides 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.

Note that the distinction between the two kinds of system description is not simply one of control flow versus data flow. Indeed, control and data relationships are present in both representations. In the first, the dependency relationship between modules determines not only which procedures call which procedures, but also how data flows between them (via parameters, return values, shared state, etc.). In the second, when one component interacts with another it is important to specify both how control passes between the parties as well as how the data flows between them. For example, to say that two components of a system interact through a pipe is to indicate not only that data is flowing from the pipe writer to the reader, but also the fact that the writer of the pipe controls how much data is placed in it and that the reader can choose not not to read the data placed in the pipe by the writer.

Although 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. For example, a blackboard system [Nii86], consisting of a central blackboard repository and a set of knowledge sources, would look quite different in the two views. A definition/use description would typically expose the implementation dependencies between modules representing knowledge sources, and the data structures that characterize the data stored in the blackboard. An architectural diagram, however, would highlight the runtime control and data paths between the blackboard and knowledge sources. For example, it would probably show which knowledge sources are registered for which blackboard

layers, and highlight the interactions between knowledge sources and the shared repository.

Given the utility of providing high level characterization of a system's structure, it is not surprising that an essential design artifact for most large systems is an architectural design that highlights the major computational components and their interactions. Unfortunately, such descriptions have the drawback that at present they remain only 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 formal basis for architectural connection.

3 Requirements for Specification 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.[2] 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. Moreover, it is a view that is shared by almost all recent work in architectural representation [Gar95, IEE95].

As a simple example, consider a system organized around 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 server. 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 architectural connection. Before presenting our approach, 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.

Expressive 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 type of 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

[2] In Section 11 we return to the issue of why it is important to distinguish between components and connectors.

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 use of any connector is compatible with its definition. 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 (strictly speaking) the pipe expects a filter on both ends.

These goals for architectural analysis can be viewed as architectural analogues of goals currently achieved at the programming language level by compilers and other program analysis tools. The goal of independent specification of connector types is analogous to the use of abstract type definitions in programming languages. The goal of checking for well-formedness is analogous to the use of type checking to guarantee that all uses of procedures are consistent with their definitions. The goal of flexibility is analogous to use of subtyping for supporting reuse. In the remainder of this paper we show how these goals can be realized for software architecture.

4 Architectural Description

We begin by describing our general approach to architectural description. This approach is embodied in the WRIGHT architectural description language, a notation that we have developed for the specification and analysis of software architectures.[3] In the next section we return to the issue of specifying connectors.

Figure 3 shows how a simple client-server system would be described using WRIGHT. The architecture of a system is described in three parts.

The first part of the description defines *component* and *connector* types. A component type is described 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.[4] In this simple example the Server and Client components both have a single port, but in general a component might have many ports. Thus ports allow a component to define multiple interfaces to other parts of a system.

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. That is they act as a *specification* that determines the obligations of each component participating in the interaction.

For example, the client-server connector type illustrated above (C-S-connector) has a client role and a server role. As we will see later, 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 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 a certain order: client requests service, server handles

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

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

```
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

request, server provides result, client gets result. (Although the connectors in this example are binary, in general a connector can have more than two roles.)

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 connector (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.

Referring to the earlier example of Figures 1 and 2, the Capitalize system would have four component types (one for each of the filters) and a single pipe connector type (of which there would be four instances). A split filter would have one input port and two output ports. A merge filter would have a two input ports and one output port. The upper and lower filters would each have a single input and output port. The attachments would be connect the system into the graph indicated in Figure 2.

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 specify the possible behaviors of each participant in an interaction, while the glue describes how these behaviors are combined. 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. (Later we will also use CSP as the semantic model to define the semantics of a Wright specification.)

While CSP has a rich set of constructs for describing communicating entities, we will use a small, but non-trivial, subset of these, including:

- **Processes and Events:** A process describes an entity that can engage in communication events.[5] 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 with which a process P can communicate 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:** ("external choice") A process that can behave like `P` or `Q`, where the choice is made by the environment, is denoted $P \,[]\, Q$. ("Environment" refers to the other processes that interact with the process.)

- **Decision:** ("internal 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.

- **Parallel Composition:** Processes can be composed using the $\|$ operator. Parallel processes may interact by jointly (synchronously) engaging in events that lie within the intersection of their alphabets. Conversely, if an event e is in the alphabet of processes P_1 and P_2, then

[5] 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. Rather, as we will see, processes are logical entities used as building blocks to specify the components and connectors of a software architecture.

P_1 can only engage in the event if P_2 can also do so. That is, the process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2, and for the events in the intersection of the processes' alphabets, both processes must agree to engage in the event. While we do not use parallel composition directly in the definition of ports, roles, or glue processes, we will use it to give semantics to the combination of them (Section 6).

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

Semantically speaking, in CSP a process, P, can be modelled as a triple (A, F, D), where A is the alphabet of P, F is a set of "failures", and D is a set of "divergences". The failures of a process are a set of pairs, each pair consisting of a trace and a set of events that the process can "refuse" to participate in after executing that trace. The divergences of a process are the set of traces of P, after which P can behave "chaotically" (essentially, with arbitrary behavior).

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

Second, we allow the introduction of scoped process names, as follows: **let** $Q =$ *process-expr* **in** R. This defines process Q to be *process-expr* within the scope of process 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 the label operator that does not label $\surd$. Additionally, we will use the symbol Σ to represent the set of all unlabeled events.

5.2 Connector Description

To describe a connector type we provide process descriptions for each of its roles and its glue. As a very simple example, consider the client-server connector introduced earlier.[6] This is how it might be written using the notation just outlined.

connector C-S-connector =
  role Client = (request!x$\rightarrow$ result?y $\rightarrow$ Client) $\sqcap$ §
  role Server = (invoke?x$\rightarrow$ return!y $\rightarrow$ Server) $\Box$ §
  glue = (Client.request?x$\rightarrow$ Service.invoke!x $\rightarrow$Service.return?y$\rightarrow$Client.result!y$\rightarrow$**glue**)
    $\Box$ §

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 external choice operator ($\Box$), 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

[6] We use simple examples to expose the central ideas. The reader should not assume that this indicates an inability of CSP to scale to realistic inter-component protocols. For example, see [Jif90] for a representative larger application of CSP to protocol definition. Also see Section 11.

the glue). In other words, the server is not allowed to terminate unless the environment in which it is operating "requests" termination.

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 internal choice operator ($\sqcap$), the choice of whether to invoke a service or to terminate is determined by the Client process.

Comparing the two roles, note that the different choice operators allow us to distinguish formally between situations in which a role is *obliged* to provide some services—the case of Server—and the situation in which a role may *choose* to take advantage of some services, but is not required to do so—the case of Client. This is an important distinction for characterizing architectural connection, since to understand an interaction it is critical to know what aspects of the behavior are *required* for a participant, and which are simply *available*.

The glue process coordinates the behavior of the two roles by indicating how the events of the roles work together. In the example above, glue allows the Client role to decide whether to invoke a service or to terminate, and then it sequences the remaining three events and their data.

The client-server example illustrates that the connector specification 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. However, even here the WRIGHT specification provides much richer facilities for describing the dynamic behavior in the use of those services, since it includes a description of sequencing and choice.

To take a more interesting (but still very simple) example, 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.[7] 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. They differ, 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, 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

[7] In this set of examples, for simplicity we ignore the data behavior of the connector. As a result, the main (non-initialization) part of the glue specification in each case is equivalent to RUN, the process that accepts all events in its alphabet. In a fuller shared data connector description, each event would have a data parameter and the glue process would be correspondingly more complex.

```
connector Shared Data1 =
    role User1 = set→User1 ⊓ get→User1 ⊓ §
    role User2 = set→User2 ⊓ get→User2 ⊓ §
    glue = User1.set→glue □ User2.set→glue
           □ User1.get→glue □ User2.get→glue □ §
```

```
connector Shared Data2 =
    role Initializer =
      let A = set→A ⊓ get→A ⊓ §
      in set→A
    role User = set→User ⊓ get→User ⊓ §
    glue = let Continue = Initializer.set→Continue
                        □ User.set→Continue
                        □ Initializer.get→Continue
                        □ User.get→Continue □ §
           in Initializer.set→Continue □ §
```

```
connector Shared Data3 =
    role Initializer =
      let A = set→A ⊓ get→A ⊓ §
      in set→A
    role User = set→User ⊓ get→User ⊓ §
    glue = let Continue = Initializer.set→Continue
                        □ User.set→Continue
                        □ Initializer.get→Continue
                        □ User.get→Continue □ §
           in Initializer.set→Continue
              □User.set→Continue □ §
```

```
connector Bogus =
    role User1 = set→User1 ⊓ get→User1 ⊓ §
    role User2 = set→User2 ⊓ get→User2 ⊓ §
    glue = let Continue = User1.set→Continue
                        □ User2.set→Continue
                        □ User1.get→Continue
                        □ User2.get→Continue □ §
           in User1.set→Continue
              □ User2.set→Continue □ §
```

Figure 4: Several Shared Data Connectors

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 somewhat more complex example, consider the problem of specifying a pipe connector type. It might appear to be a simple matter to define a pipe: it will have two roles, a writer and reader, each of which can determine when and how many times it will write or read (respectively). When done writing or reading, each closes their side of the pipe.

In fact, the writer role is just that simple. The reader role, however, is more complex. In particular, when a writer closes its end of a pipe 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.

Note that in this definition the glue plays an active role in coordinating the interactions. In particular, it causes the close event of the writer to appear as a read-eof event for the reader. (Note also that at this level of abstraction we are not directly modelling the FIFO discipline of the pipe. We'll return to this issue in Section 10.)

A more "industrial-strength" example of a connector, which we will not show here, is the U.S. Department of Defense's "High-Level Architecture for Distributed Simulations (HLA)" [GG95]. HLA is intended to support the coordination of different simulations. HLA defines a standard for the coordination of simulations through the communication of data object attributes and events. In HLA, members of a federation—the HLA term for a distributed simulation—coordinate their models of parts of the world through sharing objects of interest and the attributes that define them.

```
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

Each simulation in a federation is responsible for calculating some part of the larger simulation and broadcasting updates using the facilities of the Runtime Infrastructure (RTI). The HLA, in addition to defining data-sharing facilities, supports time-coordination and run-time management of simulation execution, checkpointing, and rollback. Thus, a key issue for this architecture is to specify and model both the interface of a federate to the rest of the simulation (a role) and the coordination services provided by the RTI (the glue of the connector). Specific federates (the components) can specify how they use the RTI's services (a port), and must demonstrate that they meet the requirements for participation in a federation.

6 Connector Semantics

In the discussion above we implicitly assumed that the roles and glue were combined in such a way that the glue coordinates the behavior of the connector's roles. We now make that notion precise, by providing a formal semantics for a connector specification. Here and elsewhere we give semantics to WRIGHT specifications by showing how they can be translated into CSP.[8]

The basic idea is to give a meaning to a connector description by treating roles as independent processes, constrained only by the glue (which serves to coordinate and interleave the events of the roles). To make this idea precise we use the CSP parallel composition operator, $\|$, for interacting processes (see Section 5.1). Specifically, we 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, ..., R_n, and glue *Glue* is the process:

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

[8]Note that by doing this we are using CSP in two ways: first, we adopt parts of CSP for the process specification parts of WRIGHT; and second, we now also use CSP as the semantic base to explain the non-CSP parts of WRIGHT.

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

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

In this definition we arrange for the glue's alphabet to be the union of all possible events (Σ) 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 (by virtue of the relabelling) and so each role can interact directly only 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 roles and glue: since all the roles and glue share the tick event, it is not possible for one of the roles to terminate successfully without the other roles (and glue) also doing so.

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. (*Cf.,* Figure 3.)

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

component DataUser =
  port DataRead = get→DataRead ⊓ §
  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 resulting system. That is, the roles act as a specifications for the ports: provided that the ports satisfy the role specifications (as defined later), they will stand in for those roles in the running system.

Thus, we define the semantics of an attached connector to be the protocol that results from the replacement of the role processes with the associated port processes. 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:

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

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. (Otherwise there would be no need to talk about replacement.)

An important motivation for allowing ports to differ from the roles they fill is that this scheme permits greater opportunities for reuse of connectors. For example, we would like to be able to attach a shared variable connector to a component such as DataUser (above), even though that

component only reads the value of the variable (and never takes advantage of the option of setting its value).

As another example, we would like 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). However, files allow both reads and writes, while an end of a pipe only exercises one of those operations. As a final example, consider a client-server connector that provides a number of different services. Clearly we should be able to connect a client that uses only a subset of those services.

But if ports need not be identical to the roles they fill, when is a port "compatible" with a role? Clearly not all ports can fill all roles. 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, and DataRead will never provide this event. As another example, for a port to read from a pipe (as specified in Figure 5), it must handle the eof marker properly. We address this issue in the next section.

8 Analyzing Architectural Descriptions

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

8.1 Compatibility (of a port with a role)

An important goal of architectural description is to answer the question of when two components can safely communicate using a particular form of interaction (such as a pipe). In terms of WRIGHT, this question appears in the form "can a given port be used in a given role?"

Informally, we would like to be able to guarantee that the 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 other words, the port process should be substitutable for the role process in such a way that the rest of the connector interaction can't detect that the role process has been replaced by the port process.

In CSP this informal notion is captured formally by means of a "refinement relationship" between processes. Formally, refinement is based on the characterization of a processes as the triple (A, F, D) of alphabet, failures and divergences (*cf.*, Section 5.1 or [Hoa85, chapter 3]). A process P is *refined by* process Q, written $\mathsf{P} \sqsubseteq \mathsf{Q}$, if their alphabets are the same, the failures of P are a subset of the failures of Q, and the divergences of P are a subset of the divergences of Q.

For the purposes of this paper, the most important property of refinement is that if $\mathsf{P} \sqsubseteq \mathsf{Q}$, then Q must respect all of P's obligations to interact with its environment. However, where P permits an internal choice among several alternatives, Q may further constrain those choices.

So, for example, suppose P and Q are processes with alphabet $\{e, f\}$:

1. if $\mathsf{P} = (e \rightarrow \mathsf{P} \sqcap f \rightarrow \mathsf{P})$ and $\mathsf{Q} = (e \rightarrow \mathsf{Q})$ then $\mathsf{P} \sqsubseteq \mathsf{Q}$
2. if $\mathsf{P} = (e \rightarrow \mathsf{P} \,[]\, f \rightarrow \mathsf{P})$ and $\mathsf{Q} = (e \rightarrow \mathsf{Q})$ then $\mathsf{P} \not\sqsubseteq \mathsf{Q}$.

In the first example, process Q does not exercise the internal choice of engaging in f, but is otherwise consistent with the behavior of P. In the second, Q is *not* a refinement of P, because it refuses f when P does not.

While CSP's notion of refinement is a good model for characterizing substitutability of ports for roles, there are two reasons why it is not possible to use CSP's definition of refinement directly to define port-role compatibility.

The first reason is a notational technicality: CSP's $\sqsubseteq$ relation assumes that the alphabets of the compared processes are the same. In general, for ports and roles this won't be the case, since a component's port may not take advantage of parts of an interaction (as when the DataRead chooses never to set a shared variable), or may offer possibilities that are not used by an interaction (as when a File offers both reading and writing, but a Pipe uses only one of these).

However, we can handle this first 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. (Formally, $P_{+B} = (P \| STOP_B)$.)

The second reason relates to methodological concerns about the ability to have a port fill as broad set of roles as possible (or equivalently to reuse a connector in as many contexts as possible). 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 *over traces described by the role.* This is because any trace of the port not described by the role represents a situation that will not arise in the interaction described by the connector.

We can capture the idea of restricting a process P to the set of traces of a process R by placing P in parallel with a process R' that has just the traces of R, but does not exercise any of R's choices. Thus any internal choices that are made by P will still be present in the combined process, except those that would have resulted in a trace not allowed by R. (Recall that the trace set of $P \| R'$ is the intersection of the trace sets of P and R.) Hence, what we desire is the process R' that is a *deterministic* version of R. Informally, R' can be obtained from R by replacing all of the non-deterministic choices from R by deterministic choices. (More on this in Section 9 where we explain how to calculate R' from R and give an example.) Formally, the deterministic version of R, $det(\mathsf{R})$, is defined from the traces of R as follows:

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 $R_{+(\alpha P \setminus \alpha R)} \sqsubseteq (P_{+(\alpha R \setminus \alpha P)} \parallel det(R))$.

Using these definitions, we see that (for example) port

$$DataRead = \text{get} \rightarrow DataRead \ \sqcap \ \S$$

is compatible with role specified by the process

$$User = \text{set} \rightarrow User \ \sqcap \ \text{get} \rightarrow User \sqcap \S$$

because the role could always decide to engage in get. On the other hand, DataRead is *not* compatible with the 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. If we have a port

$$NeedInit = \textbf{let} \ Good = \text{use} \rightarrow Good \ \Box \S \ \ \textbf{in} \ \ \text{init} \rightarrow Good \ \Box \text{use} \rightarrow \text{error} \rightarrow \text{STOP},$$

it would be compatible with role

$$GetsInit = \textbf{let} Cont = \text{use} \rightarrow Cont \ \Box \S \ \textbf{in} \ \text{init} \rightarrow Cont$$

because GetsInit indicates that an init event will occur first. The error behavior triggered by a use without an init is guaranteed never to occur by the role, and the trace restriction in the test ensures that the error behavior is not encountered.

Figure 6 provides two more examples to illustrate how this notion of compatibility supports reuse. Component DataFile supports both reading and writing on its File port. This port is compatible with both the Reader and Writer roles of a pipe shown earlier in Figure 5. So this component (or any other component with this kind of port) could be used in a system of pipes and filters.

The second example partially specifies a specialized filter, ReadOneFilter, that reads a single value from its input (ignoring all subsequent input) and then writes a single value to its output. The ReadOne and WriteOne ports of this filter are compatible with the Reader and Writer roles of a pipe connector (respectively). Hence it can be used as the source or sink of the more general pipe connector. That is, the pipe connector can be reused in this more constrained situation.

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.

```
component DataFile =
    port File = let Exit = close→ §
                in let Action = (read→Action □ write→Action)
                in Action □ Exit
    spec = ...

component ReadOneFilter =
    port ReadOne = read→close→ §
                 □ read-eof→close→ §
    port WriteOne = write→close→ §
    spec = ...
```

Figure 6: Additional Examples

An important property of any system of interacting parts is that the parts work together smoothly. This means that the assumptions that each system component makes about its environment are consistent with the other parts that form its environment in the overall system. In terms of architectural design, a key aspect of determining whether parts fit together compatibly is detecting whether the parts can get stuck during their interactions. By "get stuck" we mean that one or more parts of a system are waiting for the rest of the system to do some action that will never be forthcoming.

In terms of architectural connectors we would like to avoid the situation in which two (or more) components are waiting 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.

From a formal point of view this problem can be cast as one of detecting system deadlock. Specifically, we say 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. In other words, the roles and glue work in such a way that if the overall connector process stops, it will be in a situation that is a success state for all the parties. 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 $last(t) = \surd$.

Being able to determine whether a connector is deadlock free provides a kind of sanity check on the connector type definition. But this solves only part of the problem. Since a connector is instantiated by substituting specific ports for the connector roles, we would also like to be able to tell whether the result remains deadlock-free after having performed that substitution. 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 are identical to their respective 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 substitutability properties of refinement: the failures of a refinement must 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 (see Definition 4). 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 is of the form "$(R_1.good \rightarrow \S) \,\Box\, (R_1.crash \rightarrow \text{STOP})$." If the glue ever engages in the $R_1.crash$ event, then the connector will deadlock (by becoming the unsuccessfully terminated STOP process). However, suppose the event *crash* is not permitted by role R_1— let's say R_1's behavior is "$good \rightarrow \S$." 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 allows the *crash* behavior (*e.g.*, "$(good \rightarrow \S) \,\Box\, (crash \rightarrow \text{STOP})$"). 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. In particular, we will require that the glue be responsible for preventing behaviors that are not covered by the role specifications.

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 $\text{C} = \mathit{Glue} \parallel (\text{R}_1{:}r_1 \parallel \text{R}_2{:}r_2 \parallel \ldots \parallel \text{R}_n{:}r_n)$ is *conservative* if $\mathit{traces}(\mathit{Glue}) \subseteq \mathit{traces}(\text{R}_1{:}r_1 \| \text{R}_2{:}r_2 \| \ldots \| \text{R}_n{:}r_n)$.

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

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

$$
\begin{array}{lll}
 & \alpha(\mathrm{R}_1{:}P_1 \| \ldots \| \mathrm{R}_n{:}P_n) \subseteq \alpha Glue & [\text{definition of } \alpha Glue] \\
(1) & \quad \Rightarrow t \in traces(Glue) & \\
(2) & \quad traces(Glue) \subseteq traces(\mathrm{R}_1{:}R_1 \| \ldots \| \mathrm{R}_n{:}R_n) & [C \text{ conservative}] \\
 & (1) \wedge (2) & \\
 & \quad \Rightarrow t \in traces(\mathrm{R}_1{:}R_1 \| \ldots \| \mathrm{R}_n{:}R_n) & \\
 & \quad \Rightarrow t \in traces(C) & [\text{semantics of } \|] \\
(3) & \quad \Rightarrow (t, \alpha C) \notin failures(C) & [C \text{ is deadlock-free, } last(t) = \surd]
\end{array}
$$

$$
\begin{array}{lll}
 & R_i \sqsubseteq P_i \| det(R_i) & [P_i \textbf{ compat } R_i] \\
 & C = Glue \| \mathrm{R}_1{:}R_1 \| \ldots \| \mathrm{R}_n{:}R_n & [\text{definition of C}] \\
 & \quad \sqsubseteq Glue \| \mathrm{R}_1{:}(P_1 \| det(R_1)) \| \ldots \| \mathrm{R}_n{:}(P_n \| det(R_n)) & [\|, L : \text{monotonic}] \\
 & \quad = Glue \| \mathrm{R}_1{:}det(R_1) \| \ldots \| \mathrm{R}_n{:}det(R_n) \| \mathrm{R}_1{:}P_1 \| \ldots \| \mathrm{R}_n{:}P_n & \\
 & & [\| \text{ symmetric, } L : \text{distributes over } \|] \\
 & \quad = Glue \| \mathrm{R}_1{:}P_1 \| \ldots \| \mathrm{R}_n{:}P_n & [C \text{ conservative, lemma 1)}] \\
 & \quad = C' & [\text{definition of } C'] \\
 & C \sqsubseteq C' & \\
 & \quad \Rightarrow failures(C') \subseteq failures(C) & [\text{definitions of } \sqsubseteq] \\
(4) & \quad \Rightarrow (t, \alpha C) \in failures(C) & [(t, \alpha C) \in failures(C')]
\end{array}
$$

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

9 Automating Compatibility Checking

An important motivation for this work is the potential for automating the analysis of architectural descriptions. In particular, we would like to be able to use tools to determine whether connectors are well-formed (deadlock-free and conservative), as well as whether ports are compatible with their roles.

Thus, 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 tools for exhaustive state exploration (such as model checkers) [BCM+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 can check the compatibility of *DataRead* with the role *User* using FDR [FDR92], a commercial tool that can 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:

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

These are encoded in FDR[9] as:

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

According to Definition 4, to test the compatibility of *DataRead* with *User*, we must determine whether

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

The first part of the translation requires us to augment the alphabet of *User*, constructing a process $USERplus = User_{+(\alpha DataRead \setminus \alpha User)}$. However, because $\alpha DataRead \subseteq \alpha User$, it follows that $User_{+(\alpha DataRead \setminus \alpha User)} = User_{+\{\}} = User$:

```
USERplus = USER
```

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

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

Next we encode $det(User)$. To do this, we change the internal $\sqcap$ to the external $\Box$:

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

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

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

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

The translation process that we have just illustrated is relatively straightforward to automate. We have done so with using the a Synthesizer Generator structure editor for Wright as a front end, and implementing the transformations using the attribute grammars provided by the tool [RT89]. The only non-trivial part is calculating $det(P)$ for a process P.[10]

Roughly, the construction of $det(P)$ is carried out by replacing all occurrences of $\sqcap$ with $\Box$ as we have shown in our example. This simple construction is complicated by the fact that a process $P \Box Q$ can be on-deterministic if P and Q accept a common set of events. For example, "$P = a \rightarrow b \rightarrow \text{STOP} \Box a \rightarrow c \rightarrow \text{STOP}$" is non-deterministic: after the initial event a, one of b or c will be accepted, but not both. In this case, $det(P) = a \rightarrow (b \rightarrow \text{STOP} \Box c \rightarrow \text{STOP})$. Thus, we must construct a process so that whenever the initial events of P and Q differ in $P \Box Q$, the process is restructured as an external choice "$e \rightarrow (P(e) \Box Q(e)) \Box f \rightarrow (P(f) \Box Q(f))...$"

To complete the analysis for compatibility FDR is simply given the command:

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

This causes FDR to check whether *USERplus* is refined by *DATAREADpD*. If it is not, FDR will print a counterexample, which indicates what trace of events led to a discrepancy between the two processes (*i.e.*, a trace that could cause one process to deadlock, but not the other).

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. Similarly, deadlock-freedom can be expressed as a refinement check of the most nondeterministic deadlock-free process. For example, the most non-deterministic deadlock-free process over $\{e, f\}$ is "$DFEF = (e \rightarrow DFEF) \sqcap (f \rightarrow DFEF) \sqcap \S$." Any deadlock-free process with the alphabet $\{e, f\}$ is a refinement of $DFEF$.

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. While we might have spotted this problem for the simple examples of this paper, for more complex systems, the problems may be quite difficult to detect using informal means.

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.

[10] Although a tool like FDR maintains such information internally, the version of the checker that we are using does not currently make it available to the tool user.

```
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 7: A Pipe Connector Augmented with a Trace Specification

But, just as traditional type declarations capture only part of the "semantics" of a module, a finite state protocol typically only approximates 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. A trace specification is a predicate that must hold for every trace of the glue.

The properties of the trace are described with the following special notations: e_n represents the *nth* instance of event e in the trace. e without a subscript means that the event e occurs somewhere in the trace. $\#e$ is the number of occurrences of the event e in the trace. Thus in the example shown in Figure 5, "*Reader.read-eof* $\Rightarrow$ (*Writer.close* $\wedge$ #Reader.read = #Writer.write)" means that for every trace in which *Reader.read-eof* occurs, there must also be an occurrence of the event *Writer.close*, and the number of times that *Reader.read* has occurred equals the number of occurrences of *Writer.write*. (That is, before *eof* is signalled, all data has been read and the pipe is closed.)

A trace predicate is generalized to cover every instance of an event in a trace by $\forall\, e_i.p \bullet pred(i,p)$. This expression indicates that for every event e in the trace, $pred(i,p)$ must hold, where i is the sequence number of the event and p is the data value passed with that occurrence of the event. Similarly, $\exists\, e_i.p \bullet pred(i,p)$ indicates that there must be an event e with sequence number i and data value p. Thus, our specification $\forall$ *Reader.read*$_i$!$y \bullet \exists$ *Writer.write*$_j$?$x \bullet i = j \wedge x = y$ indicates that for every instance of the Reader.read event there is a corresponding Writer.write event with the same data value. In other words, the reader receives the same data and in the same order as it was written.

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. Thus, 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, $Glue = \textbf{glue} \| proc(P)_{\alpha Glue}$, 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 a proof obligation to show that the augmented protocol is deadlock-free and conservative.

11 Discussion

Having presented how architectural connection can be specified and analyzed using WRIGHT, we now consider in more detail two key design issues: (1) the choice of notation and underlying semantics; and (2) the need for distinguished connector semantics.

11.1 Why CSP?

An obvious question is why use CSP to specify connector protocols? Why not use an alternative, possibly simpler, notation and model of concurrency?

We investigated several alternative approaches including several state machines models (I/O Automata [LT88], StateCharts [Har87], SMV [C+86], and SDL [Hol91]), Petri Nets [Pet77], and regular expressions. While these systems have been used to model protocols and have well-defined mechanisms for composition, we favor the use of CSP for three reasons.

First is our concern with being able to capture certain critical properties of architectural connection. These properties include the ability to characterize the dynamic behavior of inter-component communication, to specify which components are responsible for making decisions during interaction, and to detect mismatched assumptions that could cause a component to get "stuck" midway through its interaction with another component.

CSP provides an ideal semantic basis for these properties. In particular, it is the only formal notation for concurrent systems that has both external (deterministic) and internal (non-deterministic) choice operators. These operators allow us to state precisely where the responsibilities for action and reaction lie in a system. Moreover, to the extent that deadlock provides a good model for what can go wrong when components interact, CSP is provably the right semantic model, since formally, it is fully abstract with respect to deadlock freedom of communicating processes [BR85].

Second is the need for a simple but powerful form of composition. Architecture is inherently about putting parts together to make larger systems. CSP's parallel composition operator works particularly well in this regard. In particular, it has the desirable compositional property that the traces of a (parallel) composition must satisfy the the specifications of each of its parts. This means that one can reason about the behavior of a system's parts separately, confident that the resulting system will continue to respect the properties established about the parts.

Third, is the pragmatic concern for tools that can assist with automated analyses of specifications. We believe that unless we can provide for formal specifications the kinds of direct, automated feedback that compilers give current programming notations, there is little hope of getting engineers to use our notations for real systems.

This is not to argue that CSP is the *only* notation that could have been used for WRIGHT. Indeed, we recognize that the choice of CSP allows us to capture only a certain class of properties. For example, WRIGHT does not handle properties such as timing behavior of interactions, or fairness 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.

11.2 Why not CSP?

Having argued that CSP provides a particularly well-suited semantic basis for specification of architectural connection, one is tempted to ask, why not go all the way and use *just* CSP as the notation? That is, why is WRIGHT needed at all?

It is true that WRIGHT does not add formal power to CSP. However, we believe that there are important methodological reasons why it makes sense to provide a specialized notation for architectural specification.

First is the goal of elucidating the architectural abstractions: components, connectors, configurations. Each of these plays a prominent role in architectural specification as it is practiced, and so it is important to understand what purpose each serves in a specification and how they are related. By providing a notation with explicit constructs for describing these abstractions, we match the vocabulary of the architect's informal design practices. This means that it should be easier to map one into the other, both in terms of formalizing an informal architectural specification and communicating the results of architectural analysis to system developers.

In addition to the benefits of defining the architectural abstractions, regardless of how well they can be simulated using abstractions of CSP, there are three aspects of architectural specifications that make the additional notational structures of WRIGHT valuable in easing the specifiers task. These are (1) encapsulation, (2) the type/instance distinction, and (3) the asymmetrical roles of components (as context-independent behaviors) and connectors (as context-setting definitions).

As we saw earlier, a WRIGHT specification structures a system's architecture the definition by providing both local, interface specifications, such as the role specifications on a connector, and overall behavior specifications, such as the connector's glue. The role specifications localize one aspect of the interaction behavior to simplify consistency-checking and other analysis. In order to achieve this localization, the definition of a connector role (and of ports, computations, and glue, for that matter) should not depend on elements of the specification that are outside the local area of concern. The architectural specification thus encapsulates each definition so that it can be analyzed independently of the rest of the architecture, and can potentially be used in many different contexts. Because CSP uses global event names (*i.e.*, the $\|$ operator uses exact, unstructured name matching), there must be an additional level of structuring to ensure that the descriptions are not falsely combined because of name clashes, and that system parts are not prevented from being combined by not having selected the same name for the same construct. This

level of structuring, achieved here by the systematic relabelling of events, can either be redone explicitly and individually for every architectural specification using a base formalism such as CSP, or it can be provided implicitly and automatically by an architectural language such as WRIGHT.

Closely related to the goal of encapsulation is the important distinction between types and instances. A common difficulty in understanding an architectural specification is to know whether the specification represents a *type* of component or connector to be used in a class of systems, or a specific *instance* of the structure of an individual system. If we read a specification of a pipe, are we being informed of the interaction that will take place between two particular filters in a particular system, or are we trying to understand a general pattern of interaction that can occur many times in any given system? By reducing all descriptions to the same construct, the process, CSP does not help us answer this question, and so we must add this information to any specification.

An important distinction in architectural specification is between the *context-independent*, encapsulated behaviors provided by components and the *context-setting* interaction patterns provided by connectors. A result of this distinction is that the "interfaces" to the components and connectors (ports and roles, respectively), both describe the behavior of the components. Ports describe the component with which it is associated while roles constrain a component which may participate in the interaction represented by the connector. Similarly, a component's computation describes the behavior of the component independent of how it will be used in the configuration. The connector's glue, on the other hand, describes a pattern of use—that is, it describes how the components will be fit into the configuration. This distinction, between a context of use and a use independent of context, is not made in CSP: By virtue of CSP's uniform treatment of all entities as processes, all behavior descriptions are equal.

11.3 Why distinguish connectors from components?

WRIGHT's most significant point of departure from most existing approaches to system specification is its focus on connectors as explicit semantic entities. One might well question the need for both components and connectors: why not instead have only components, with connectors represented as special kinds of components?

Other architectural description languages have adopted this latter approach. For example, Rapide [LVB+92] uses events to characterize component interaction, but, unlike WRIGHT, provides a fixed set of connector types to characterize how events flow between components. To model a new kind of connector, such as a pipe, in Rapide one would create a new type of component that handles the pipe buffering.

Clearly the advantage of taking such an approach is that it simplifies the language. Moreover, if there is only one form of abstraction in the language (namely that used to define computational elements), the mapping to a semantic base (such as CSP, or, in the case of Rapide, Posets) is simpler.

However, when designing a language (of any sort) it is important to be clear about its intended purpose. If its primary function is to support reasoning and formal manipulation, then a language should generally strive for minimality. On the other hand, if the primary function is to provide a vehicle of expression that matches the intuitions and practices of users, then a language should aspire to reflect those intuitions and practices [SG95].

We view WRIGHT as fitting the latter mold. Although the ability to reason about an architectural specification is key, our first concern has been to provide a good match to the abstractions that are used by practitioners who routinely need to describe software architectures. As we argued in Section 3, the use of new abstractions for component interaction are central to architectural design. In particular, the "lines" connecting the computational elements of such a design clearly have a different status than the computational elements (the "boxes"), and further, those lines may often represent abstractions with their own non-trivial semantics.

We have attempted to provide a notation that is oriented around the explicit identification, characterization, and reasoning about those interactions. Thus, although from a logical point of view explicit connectors are not strictly necessary, from a practical point of view we feel they are indispensable. Rather than force software architects to encode their designs into a formalism that is good for theoreticians, we aim to provide them with a notation that allows a direct expression of the abstractions needed for architectural design. We are willing to pay for this convenience with a somewhat more complex mapping between the notation and the underlying semantics.

A secondary benefit from the approach that we have taken is its support for reuse. Most systems are constructed using a small set of interaction types, such as rpc, event broadcast, etc. By separating the relatively fixed specifications of connectors from the relatively variable specifications of components, WRIGHT reflects this kind of reuse. Moreover, the definition of port-role compatibility is directly responsive to connector reuse, since it permits many different kinds of component ports to be used with the same kind of connector role. For example in Figure 2, connector types (here, pipes) are reused many times. Section 8.1 illustrated this point more formally.

12 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, they use the distinction between input and output events to capture the assumption/promise dichotomy, while we use the distinction between external and internal choice.

Description of Software Architecture

A number of other architectural representation languages have been proposed. Rapide [LVB+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. As we have noted, in Rapide connectors are modeled by defining new kinds of components, whereas in WRIGHT connectors function as composition operators.

The UNICON language [SDK+95] 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.

Another closely related system is the SARA System [EFRV86]. It was developed as a method and toolset for design of system architectures for concurrent systems. SARA includes a specification language, called GMD (Graph Model of Behavior), that has goals similar to those of WRIGHT, insofar as both provide a formal basis for modelling the protocols of interaction between components of system design. The formal basis for analysis of GMD specifications is Petri Nets (via an intermediate translation stage through Transition Expressions), for which there exists considerable theory and automated tools for checking properties. The primary difference between SARA and the work described here is the use of connector types as first class, "instantiatable" entities. Because WRIGHT separates the definition of (reusable) connector types from their use in a given system, it must provide criteria for determining when a connector can be used in a given context (*i.e.*, port-role compatibility), an issue that simply doesn't arise in SARA.

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.

Specifications, Protocols, and Refinement

Traditionally, research on protocols has been concerned with developing algorithms to achieve certain communication properties—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 WRIGHT provides a specific way of structuring the description of connector protocols—namely, separation into roles and glue. As we argued earlier, 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, LM86]). That work focusses on how proofs of satisfaction of specifications can be structured through compositional proof techniques. In order to achieve this effect, they consider the question of whether refinement holds in a particular context. That is, for a process context $\mathcal{C}[\cdot]$ they define a relation $\mathsf{Q} \sqsubseteq_{\mathcal{C}} \mathsf{P}$ to hold if $\mathcal{C}[\mathsf{Q}] \sqsubseteq \mathcal{C}[\mathsf{P}]$. In our work, we have used relaxed refinement in a slightly different way, because we use the role process R not just to define the "specification" process in the contextualized-refinement test, but also the context. In the notation above, the consistency test would be roughly $\mathsf{R} \sqsubseteq_{\mathcal{C}} \mathsf{P}$ where $\mathcal{C}[\cdot] = \cdot \| det(\mathsf{R})$. Thus, the context is constructed from the processes to be compared, rather than being independent of those processes.

Another related use of protocols is in work on extending object-oriented systems with protocols over an object's methods. For example, 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 [YS94] 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.

13 On-going and Future Research

In this paper we have described an approach to specifying architectural connection. While it is a good first step toward a more rigorous discipline for architectural design, there are clearly many issues that this work does not directly address. Several of these are topics of current work by the authors; many are also being addressed by others in the research community.

Component Specification

In WRIGHT, ports allow one to specify the interfaces of architectural components. However, in this paper we have not addressed the specification of the actual computations performed by the components. While there are numerous notations that might be used, the obvious choice for WRIGHT is to use a CSP process description for this purpose.

The interesting issue that must then be addressed is the relationship that should exist between a port specification and a specification of a component's actual behavior. The obvious relation is that of *projection*: a port must be a projection of the full computation into the events that are related to that specific interaction. However, this is complicated by the fact that a port provides a specification not only of the component's behavior, but also of the assumptions that the component makes about the behavior of other participants in the interaction. These assumptions are indicated in its specification by the events that are in the port protocol but are not controlled by the component (*e.g.*, where two events are offered by external choice). Thus, the test of consistency between a port of a component and the component itself must take into account not just the component's computation, but also any assumptions that the component makes about each of its various interactions. Details of this test, however, are beyond the scope of this paper.

Hierarchy

We have not addressed issues of hierarchical description in this paper. Clearly any scalable architectural specification language will need to allow encapsulation of subsystems as components (or connectors) in other systems. The key issue to resolve is what should be the "correctness" relationship between a subsystem and the architectural element that it represents.

Within the context of WRIGHT there is an obvious answer to the question: the subsystem must be a refinement of the element it represents, once events internal to the subsystem have been hidden. That is, a subsystem should be substitutable for the component or connector that it represents. For finite WRIGHT specifications this property is checkable using automated tools.

Dynamism

CSP is inherently limited to systems with static process structure. That is, the set of possible processes must be known at system definition time: new processes cannot be created or passed as parameters in a running system. WRIGHT inherits this limitation. However, it is clear that there are many architectures that are fundamentally dynamic. For example, one might be developing an air traffic control system in which airplanes become connected to the nearest tracking station as they fly through space.

Finding an appropriate formal basis for dynamic architectures remains an open and active research topic. Some progress has been made by others. Rapide allows the dynamic creation of components, much as traditional object-oriented system can dynamically create new instances of objects. Darwin [MDEK95] uses the Pi Calculus [MPW92] as its basis, a formalism specifically designed to handle mobile processes. Architectural dynamism has also been modelled using the Chemical Abstract Machine as the underlying formalism [IW95].

Style

An important property of many architectural descriptions is that they conform to a set of design rules that allow specialized analyses to be performed. For example, a signal processing system might be written using acyclic graphs of pipes and filters to enable direct calculation of system throughput and latency.

Such sets of design rules are sometimes called architectural styles. In particular, stylistic rules typically determine a vocabulary of component and connector types, topological constraints, and constraints on the nature of the computations that can performed [AAG95].

As presented in this paper, WRIGHT goes part way towards defining styles by allowing the definition of component and connector types. But it does not allow one to state topological constraints or other global, semantically-based constraints. The specification of architectural styles is an active area of research, and extensions of WRIGHT to support style definition are currently being developed as part of one of the author's Ph.D. thesis.

14 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 application of formal machinery (through CSP) 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.

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. The anonymous reviewers for TOSEM also provided many excellent suggestions for improvements.

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.

References

[AAG95] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[BR85] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *Proceedings NSF-SERC Seminar on Concurrency*. Springer Verlag, Lecture notes in Computers Science, 1985.

[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] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden Vallen, PA, July 1990. Software Engineering Institute.

[EFRV86] Gerald Estrin, Robert S. Fenchell, Rami R. Razouk, and Mary K. Vernon. Sara (system architects apprentice): Modelling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[FDR92] *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, 1.2β edition, October 1992.

[Gar95] David Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995. Published as CMU Technical Report CMU-CS-95-151, April 1995.

[GG95] Peter Green and Terry Griffin. Specification for the RTIS HLA/RTI implementation. Technical Report RTIS10951, The Real-Time Intelligent Systems Corporation, Westborough, MA, October 1995.

[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.

[IEE95] Special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[IW95] Paola Inverardi and Alex Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):373–386, April 1995.

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

[LM86] Kim G. Larsen and Robin Milner. A complete protocol verification using relativized bisumulation. Technical Report ECS-LFCS-86-13, University of Edinburgh Laboratory for Foundations of Computer Science, September 1986.

[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.

[LVB+92] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent timed systems, March 1992.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[MG92] Erik Mettala and Marc H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA '93*, volume 28 of *ACM Sigplan Notices*, pages 1–15, October 1993.

[Nii86] H. Penny Nii. Blackboard systems Parts 1 & 2. *AI Magazine*, 7 nos 3 (pp. 38-53) and 4 (pp. 62-69), 1986.

[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 *Proceedings of the Ninth International Conference on Software Engineering*, pages 61–68, Monterey, CA, March 1987. IEEE Computer Society Press.

[Pet77] J.L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[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):57–66, July 1990.

[RT89] Tom Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

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

[SG95] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Volume 1000. Springer-Verlag, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

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

[YS94] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *Proceedings of OOPSLA'94*, October 1994.