Formalizing Architectural Connection*

16th International Conference on Software Engineering, Sorrento, Italy, May, 1994

Robert Allen David Garlan

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Abstract

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

1 Introduction

As software systems become more complex the overall system structure – or software architecture – becomes a central design problem. Design issues at this level include gross organization and control structure, assignment of functionality to computational units, and high-level interactions between these units. While the design of a good software architecture has always been a significant factor in the success of any large software system, it is only recently that the specific topic of software architecture has been identified as a focus for research and development in workshops [?DesignWorkshop93, ?Dagstuhl93], government funding and industrial development [?Mettala92].

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

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

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

In this paper we describe a first step towards these goals by providing a formal basis for specifying the interactions between architectural components. The essence of our approach is to provide a notation and

*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.

underlying theory that gives architectural connection explicit semantic status. Specifically, we provide a formal system for specifying architectural *connector types*. The description of these connector types is based on the idea of adapting communications protocols to the description of component interactions in a software architecture.

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

We begin by describing a general framework for architectural description and then briefly characterize the hard problems in developing a theory of architectural connection. Next we outline our notation and illustrate through examples how it is used to solve these problems. Having motivated the approach we provide a formal semantics and show how this leads to a sound deductive system in which architectural compatibility can be checked in a way analogous to type checking in programming languages. Finally, we show how this work is related to other approaches to architectural description.

2 Requirements for a theory of architectural connection

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

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

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

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

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

```
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 1: A Simple Client-Server System

of checks to be automatable. Third, while mismatches should be detectable, we would also like to allow flexibility. For example, (as in Unix) we might want to connect a file to a filter through a pipe, even though the pipe expects a filter on both ends. Hence, independent specification is analogous to type definitions for programming languages; checking for well-formedness is analogous to the use of type checking to guarantee that all uses of procedures are consistent with their definitions; requirements of flexibility are analogous to subtyping. In the remainder of this paper we show how these goals can be realized.

3 Architectural description

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

Figure **??** shows how a simple client-server system would be described in the WRIGHT architectural description language. The architecture of a system is described in three parts. The first part of the description defines the *component* and *connector* types. A component type is described (for the purposes of this paper) as a set of *ports* and a *component-spec* that specifies its function. Each port defines a logical point of interaction between the component and its environment.[1] In this simple example Server and Client components both have a single port, but in general a component might might have more.

A connector type is defined by a set of *roles* and a *glue* specification. The roles describe the expected local behavior of each of the interacting parties. For example, the client-server connector illustrated above has a client role and a server role. Although not shown in the figure, the client role might describe the client's behavior as a sequence of alternating requests for service and receipts of the results. The server role might describe the server's behavior as the alternate handling of requests and return of results. The glue specification describes how the activities of the client and server roles are coordinated. It would say that the activities must be sequenced in the order: client requests service, server handles request, server provides result, client gets result.

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

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

4 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.

4.1 Process notation

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

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

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 [?^{CSPBook}] to define the protocols of the roles, ports and glue. (In what follows, we will assume that the reader has some familiarity with CSP.)

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

- **Processes and Events:** A process describes an entity that can engage in communication events.[2] Events may be primitive or they can have associated data (as in e?x and e!x, representing input and output of data, respectively). The simplest process, STOP, is one that engages in no events. The event $\surd$ is used represent the "success" event. The set of events that a process, P, understands is termed the "alphabet of P," or αP.

- **Prefixing:** A process that engages in event e and then becomes process P is denoted $e \to P$.

- **Alternative:** ("deterministic choice") A process that can behave like P or Q, where the choice is made by the environment, is denoted $P \mathbin{\Box} Q$. ("Environment" refers to the other processes that interact with the process.)

- **Decision:** ("non-deterministic choice") A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.

- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Unlike CSP, however, we restrict the syntax so that only a finite number of process names can be introduced. We do not permit, for example, names of the form $Name_i$, where i can range over the positive numbers.

In process expressions $\to$ associates to the right and binds tighter than either $\Box$ or $\sqcap$. So $e \to f \to P \mathbin{\Box} g \to Q$ is equivalent to $(e \to (f \to P)) \mathbin{\Box} (g \to Q)$.

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

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

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

This subset of CSP defines processes that are essentially finite state. It provides sequencing, alternation, and repetition, together with deterministic and non-deterministic event transitions.

4.2 Connector description

To describe a connector type we simply provide process descriptions for each of its roles and its glue. As a very simple example, consider the client-server connector introduced earlier.[3] This is how it might be written using the notation just outlined.

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

The Server role describes the communication behavior of the server. It is defined as a process that repeatedly accepts an invocation and then returns; or it can terminate with success instead of being invoked. Because we use the alternative operator ($\Box$), 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 the glue).

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

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

[3] We use simple examples in order to expose the central ideas. The reader should not assume that this indicates an inability to scale to realistic inter-component protocols. For example, see [?^{Jifeng90}] for a representative larger application of CSP to protocol definition.

to distinguish formally between situations in which a given role is *obliged* to provide some services – the case of Server – and the situation where it may take advantage of some services if it chooses to do so – the case of Client.

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

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

Figure ?? illustrates four possible specifications.[4] The first, Shared Data_1, indicates that the data does not require an explicit initialization value. The second, Shared Data_2, indicates that there is a distinguished role Initializer that must supply the initial value. The third alternative, Shared Data_3 is similar to the second in that it has an explicit Initializer role, but it does not require that the other participant wait for that initialization to proceed. The final alternative, Bogus, seems reasonable – the connector requires that one of the participants initialize the variable, but does not specify which one. However, if each participant proceeds, 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 ??.

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

$$
\begin{array}{l}
\textbf{connector}\ \text{Shared Data}_1 = \\
\quad \textbf{role}\ \text{User}_1 = \text{set}\rightarrow\text{User}_1 \sqcap \text{get}\rightarrow\text{User}_1 \sqcap \surd \\
\quad \textbf{role}\ \text{User}_2 = \text{set}\rightarrow\text{User}_2 \sqcap \text{get}\rightarrow\text{User}_2 \sqcap \surd \\
\quad \textbf{glue} = \text{User}_1.\text{set}\rightarrow\textbf{glue}\ \square\ \text{User}_2.\text{set}\rightarrow\textbf{glue} \\
\qquad\qquad \square\ \text{User}_1.\text{get}\rightarrow\textbf{glue}\ \square\ \text{User}_2.\text{get}\rightarrow\textbf{glue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \text{Shared Data}_2 = \\
\quad \textbf{role}\ \text{Initializer} = \\
\qquad \textbf{let}\ \text{A} = \text{set}\rightarrow\text{A} \sqcap \text{get}\rightarrow\text{A} \sqcap \surd \\
\qquad \textbf{in}\ \text{set}\rightarrow\text{A} \\
\quad \textbf{role}\ \text{User} = \text{set}\rightarrow\text{User} \sqcap \text{get}\rightarrow\text{User} \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \text{Continue} = \text{Initializer.set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User.set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{Initializer.get}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User.get}\rightarrow\text{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \text{Initializer.set}\rightarrow\text{Continue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \text{Shared Data}_3 = \\
\quad \textbf{role}\ \text{Initializer} = \\
\qquad \textbf{let}\ \text{A} = \text{set}\rightarrow\text{A} \sqcap \text{get}\rightarrow\text{A} \sqcap \surd \\
\qquad \textbf{in}\ \text{set}\rightarrow\text{A} \\
\quad \textbf{role}\ \text{User} = \text{set}\rightarrow\text{User} \sqcap \text{get}\rightarrow\text{User} \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \text{Continue} = \text{Initializer.set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User.set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{Initializer.get}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User.get}\rightarrow\text{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \text{Initializer.set}\rightarrow\text{Continue} \\
\qquad\qquad\quad \square\text{User.set}\rightarrow\text{Continue}\ \square\ \surd
\end{array}
$$

$$
\begin{array}{l}
\textbf{connector}\ \text{Bogus} = \\
\quad \textbf{role}\ \text{User}_1 = \text{set}\rightarrow\text{User}_1 \sqcap \text{get}\rightarrow\text{User}_1 \sqcap \surd \\
\quad \textbf{role}\ \text{User}_2 = \text{set}\rightarrow\text{User}_2 \sqcap \text{get}\rightarrow\text{User}_2 \sqcap \surd \\
\quad \textbf{glue} = \textbf{let}\ \text{Continue} = \text{User}_1.\text{set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User}_2.\text{set}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User}_1.\text{get}\rightarrow\text{Continue} \\
\qquad\qquad\qquad\qquad \square\ \text{User}_2.\text{get}\rightarrow\text{Continue}\ \square\ \surd \\
\qquad\qquad \textbf{in}\ \text{User}_1.\text{set}\rightarrow\text{Continue} \\
\qquad\qquad\quad \square\ \text{User}_2.\text{set}\rightarrow\text{Continue}\ \square\ \surd
\end{array}
$$

Figure 2: Several Shared Data Connectors

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

connector Pipe =
  role Writer = write$\rightarrow$Writer $\sqcap$ close$\rightarrow$ $\surd$
  role Reader =
    let ExitOnly = close$\rightarrow$ $\surd$
    in let DoRead = (read$\rightarrow$Reader
      $\square$ read-eof$\rightarrow$ExitOnly)
    in DoRead $\sqcap$ ExitOnly
  glue = **let** ReadOnly = Reader.read$\rightarrow$ReadOnly
      $\square$ Reader.read-eof
        $\rightarrow$Reader.close $\rightarrow$ $\surd$
      $\square$ Reader.close$\rightarrow$ $\surd$
    in let WriteOnly = Writer.write$\rightarrow$WriteOnly
      $\square$ Writer.close$\rightarrow$ $\surd$
    in Writer.write$\rightarrow$**glue**
      $\square$ Reader.read$\rightarrow$**glue**
      $\square$ Writer.close$\rightarrow$ReadOnly
      $\square$ Reader.close$\rightarrow$WriteOnly

Figure 3: A Pipe Connector

5 Connector semantics

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

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

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

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

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

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

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

6 Ports and connector instantiation

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

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

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

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

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

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

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

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

7 Analyzing architectural descriptions

We now consider the kinds of analysis and checking that are made possible by our connector notation and formalism. Because of space considerations we will limit ourselves to summarizing the main results: details can be found elsewhere [?ConnectorsTR].

7.1 Compatibility (of a port with a role)

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

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

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

However, it is not possible to use CSP's definition of refinement directly to define port-role compatibility for two reasons. The first is the technicality that CSP's $\sqsubseteq$ relation assumes that the alphabets of the compared processes are the same. We can handle this problem simply by augmenting the alphabets of the port and role processes so that they are identical. This is easily accomplished using the CSP operator for extending alphabets of processes: P_{+B} extends the alphabet of process P by the set B.[5]

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

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

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

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

Definition 3 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 port $\mathit{DataRead} = \text{get} \rightarrow \mathit{DataRead} \sqcap \checkmark$ is compatible with role $\mathit{User} = \text{set} \rightarrow \mathit{User} \sqcap \text{get} \rightarrow \mathit{User} \sqcap \checkmark$ because the role could always decide to engage in get. On the other hand, *DataRead* is *not* compatible with role $\mathit{Initializer} = \textbf{let } \mathit{Continue} = \ldots \textbf{in } \text{set} \rightarrow \mathit{Continue}$ because this latter role indicates that at least initially, set must be offered to the connector.

7.2 Deadlock freedom

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

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

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

Definition 4 A connector C is *deadlock-free* if for all $(t, ref) \in failures(C)$ such that $ref = \alpha C$, then $last(t) = \surd$.

By definition of port instantiation, a deadlock-free connector will remain deadlock-free when its roles are instantiated with ports that exactly match the roles. However, we would like to be able to make a stronger claim: namely, that consistency of a connector with its roles implies consistency with *any compatible* ports. To achieve this we need to define another property of connectors that guarantees the glue process does not engage in behavior outside the union of the behaviors of its roles. This will restrict the behavior of an instantiated connector to those behaviors that are covered by port-role compatibility. We need only restrict the traces for two reasons. First, CSP guarantees that the glue *must* refuse any event in its alphabet that does not continue a valid trace, and second, definition ?? ensures that all possible events are indeed in the glue's alphabet.

Definition 5 A connector C = $Glue \parallel (\mathrm{R}_1{:}r_1 \parallel \mathrm{R}_2{:}r_2 \parallel \ldots \parallel \mathrm{R}_n{:}r_n)$ is *conservative* if $traces(Glue) \subseteq traces(\mathrm{R}_1{:}r_1 \| \mathrm{R}_2{:}r_2 \| \ldots \| \mathrm{R}_n{:}r_n)$.

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

The proof of this theorem relies on the monotonicity properties of $\sqsubseteq$ and on the context providing properties of making the glue conservative.

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.

8 Automating compatibility checking

As we have indicated, an important motivation for this work is the potential for automating compatibility checks. To achieve this, we have constrained our use of the CSP notation in two ways. First, we have restricted the notation such that our processes will always be finite. (Of course, infinite traces are still possible even though we can't create an infinite number of processes.) This means that we can use Model Checking technology [?^{Burch90}] to verify properties of the processes and to check relationships between processes. Second, we have expressed our checks as refinement tests on simple functions of the described processes. In other words, we can express our tests as checks of the predicate $P \sqsubseteq Q$ for appropriately constructed finite processes P and Q. This permits us to apply the emerging technology of automated verification tools to make these checks.

To illustrate, we show how we would check the compatibility of *DataRead* with the role *User* using FDR [?^{FDR92}], a commercial tool designed to check refinement conditions for finite CSP processes. First, to use FDR we must translate our notation to fit the variant of CSP used in this tool. Recall that in our notation the processes are:

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

These are encoded in FDR[6] as:

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

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

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

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

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

```
USERplus = USER
```

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

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

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

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

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

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

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

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

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

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

9 Comparisons to other approaches

Description of software architecture:

One approach to architectural description is to use the facilities of a modular programming language. Architectural components are represented by modules (or, in some cases, objects) for which interfaces define the functionality of the component in terms of the operations it provides to the system. "Interactions" between modules are determined by name matching and the use of "imports" clauses. While the modularization facilities of programming languages may be adequate for structuring the code of a system, we believe they are not a general solution to the problem of describing software architectures. The main problem is that they provide only a limited number of interaction mechanisms – typically, procedure calls and data sharing. As a result, the designer is forced to encode abstract interactions between architectural components in terms of these facilities [?AllenIDL94].

Recent research on module interconnection has introduced a number of new mechanisms and richer notions of module interconnection [?Perry87, ?Purtilo90, ?Reiss90Field]. 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).

A number of other architectural representation languages have been proposed. Rapide [?Luckham92a] 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. Other domain-specific architectural description languages have been proposed [?Mettala92, ?DSSAworkshop]. These latter languages increase their analytic and expressive leverage by specializing to a particular family of systems.

Other formal models:

Other models of concurrency could have been used to define the semantics of connectors, including state machines, pre- and post-conditions, and Petri nets. We investigated several state machine approaches such as I/O Automata [?Lynch88], StateCharts [?Harel87], SMV [?Clarke86], and SDL [?Holzmann91]. While these systems have been used to model protocols and have well-defined mechanisms for composition, we favored the use of CSP for three reasons. First, it has a semantic basis (in terms of traces, divergences, and refusals) that makes it ideal for characterizing problems of connector deadlock and for expressing port-role compatibility as a kind of refinement. Second, it provides a powerful calculus for composing systems in terms of parallel composition. Finally, it has industrial-strength tools (such as FDR) for automated analysis.

Our choice of a "finite" subset of CSP allows us to automate checking of certain properties of architecture, but it also limits the expressiveness of the notation. It is possible to augment our notation with other kinds of formal description. Indeed, the full version of Wright [?ConnectorsTR] allows the use of auxiliary trace specifications to characterize non-finite properties. For example, a pipe connector might have a trace specification that asserts that data transmission follows a FIFO discipline, even though this is not directly expressible in our subset of CSP. Of course, the auxiliary specifications are not automatically checkable. There are other properties, however, (such as timing behavior of interactions) that we cannot handle because CSP's semantic model is not rich enough. To address such properties, one can imagine retaining the general descriptive framework (of ports, roles, and glue) for connectors, but replacing CSP with an alternative formalism.

Refinement of protocols:

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

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

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

There has been some work that, like ours, exploits the fact that in a constrained situation, the criteria of refinement can be weakened without compromising substitutability (*e.g.*, [?[Jacob87]]). However, that work uses refinement to indicate *substitution* of one process for another, in contrast to our work, in which the role serves as a *specification* for the properties of the port (and hence it *defines* the context in which it may be used as well as the properties that the port must have).

A final example of a use of protocols which relates to ours is work by Nierstrasz on extending object-oriented notations to permit specification of object types in terms of protocols over the services that they provide [?[Nierstrasz93]]. Nierstrasz extends the 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.

10 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 development of formal machinery for automatable compatibility checking of architectural descriptions, thereby making many of the benefits of module interface checking available to designers of software architectures.

In developing this basis for connectors we have adapted the more general theory of process algebras and shown how it can be specialized to the specific 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. Moreover, although we have focused only on simple examples in this paper, the notation can be used to define a wide variety of rich architectural interactions.

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

Acknowledgements

We would like to thank Gregory Abowd, Stephen Brookes, Jose Galmes, Daniel Jackson, Elliot Moss, John Ockerbloom, Mary Shaw, Scott Vorthmann, Jeannette Wing, and the ICSE reviewers for their constructive comments on this research.