Using Refinement to Understand Architectural Connection

David Garlan
Department of Computer Science
5000 Forbes Avenue
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The predominant use of refinement is to relate specifications of a system at two levels of abstraction. In this paper we describe a different application of refinement. We consider the problem of specifying reusable architectural connectors and the associated need to have formal rules for instantiating them for a specific system. We show that it is possible to use notations like CSP for these specifications and then to adapt the notion of process refinement to provide the rules for instantiation. We further show that these rules are sound with respect to deadlock freedom.

1 Introduction

The predominant use of refinement is to relate specifications of a system at two levels of abstraction. Typically an abstraction of a system is made more concrete in a lower-level specification that is closer to an implementation. In the extreme, the lower-level specification is some kind of machine-executable language.

For most systems of refinement a set of refinement rules provide the formal basis for deciding when one description is a legal refinement of another. The general idea behind all of these rules is that the lower-level description must have behavior that is consistent with the promised behavior of the more abstract description, but that it is free to make specific choices where the higher-level description has left that choice open.

While the use of refinement for developing correct implementations is certainly a good application of this general idea, it is not the only one. In this paper we illustrate a quite different application. We consider the problem of specifying reusable architectural connectors and the associated need to have formal rules for instantiating them in a specific system. As we will show, it is possible to use notations like CSP for these specifications and then to adapt the notion of process refinement to provide the rules for instantiation. We further show that the choice of refinement is not only intuitively appealing for this application, but also allows us to prove that certain important properties of the connector are maintained at the point of instantiation.

In the remainder of this paper we first outline the problem that motivates this work. Next we show how connector types can be defined as a collection of interacting protocols written in a language like CSP. Then we consider the problem of instantiating these connectors and show how process refinement

can be adapted as a solution. Finally, we show that this notion of refinement allows us to guarantee preservation of deadlock freedom in the presence of instantiation.

2 Architectural Specification

For large systems the overall system structure – or software architecture – becomes a critical design problem. Most systems of any size typically are presented in terms of a set of high-level interacting components. For example, a management information application may consist of a central database accessed by a set of applications which are accessed through a shared user interface.

The ubiquitous use of architectural concepts is highlighted by the typical informal documentation associated with system description. Usually a system is pictured as a boxes and lines diagram in which the boxes represent the main computational components and the lines represent interactions between those components. The prose that accompanies these figures uses phrases like "pipe and filter system", "client-server organization", "blackboard architecture", and "layered organization" to describe common idiomatic architectural patterns – or architectural styles [5].

For some domains architectural conventions have become standardized in a way that permits descriptions of architecture in terms of specific components and connectors. We are all familiar with the canonical architecture for a compiler. But other application-specific architectures (sometimes called "reference architectures") are becoming increasingly important to industry as a vehicle for design and code reuse, interoperability, standardization, and automated development support [8, 7, 4]. These systems gain their power by exploiting a set of design constraints and conventions that dictate such things as global system organization, and provide a common vocabulary of design elements (such as parsers and protocol layers).

While architectural description is crucial for large-scale software development, the relative informality of most architectural descriptions seriously limits their utility. It is often difficult to know precisely what is meant by terms such as "client-server". It is usually impossible to analyze the descriptions or to infer non-trivial properties from them. It is impossible to compare different architectural alternatives. It is hard to check that an implementation respects the constraints implied by an architectural description.

What is needed is a way to specify software architectures. Such a form of specification should allow a natural mapping of the informal notions into a more precise notation. In particular, it should be able to give meanings to boxes and lines diagrams and account for the idiomatic uses of architectural terms. It should also permit the designer to reuse general concepts from one architectural description to another. Further, it should allow one to check whether an architectural description is consistent, in the sense that the parts work well together.

It is important to note that this is not simply a problem of being able to specify a given system at a high level of abstraction. Rather, what is required is a building block approach to system specification. Concepts like "client-server" and "pipe connection" should become reusable specifications that can be incorporated into specific systems. This requirement is crucial to support the

definition of architectural styles and reference architectures, which allow new products to be designed around a common vocabulary and set of conventions about system organization.

But the approach advocated here raises a number of fundamental questions: What does it mean to specify a reusable architectural building block? What does it mean to use (or "instantiate") one of these? What are the rules for checking that a use is consistent with its definition? What significant properties do these checks guarantee? In the following sections we provide partial answers to these questions.

3 The Wright Architectural Description Language

The approach that we will adopt is the following. We view the architecture of a system as a configuration of *components* and *connectors*. A component is the locus of computation, while a connector describes the interactions that can take place between a set of components. Components have a set of interaction points, or *ports* through which they interact with their environment. Connectors link the ports of two or more connectors. To specify a system we first define a set of component and connector *types*.[1] Second, we declare instances of these types and indicate how they are combined in a bipartite graph.

To make this concrete, we have developed the WRIGHT architectural description language for describing software architectures.[2] Figure 1 illustrates how a simple client-server system would be described in WRIGHT. In this system there are two component types: Client and Server. Here each components has a single port (although, in general a component might have many ports). Additionally with each component type we provide a component specification that specifies its function. (For the purposes of this paper, we will not concern ourselves with this specification).

In the figure we also declare a single connector type. 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. In the above example, the client-server connector has a client role and a server role. As we will soon see, 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 figure also includes a declaration of a set of component and connector *instances*. These define 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`).

To provide a system definition, component and connector instances are combined by indicating which component ports are attached as (or instanti-

[1] Or better yet, import them from an existing library of architectural elements.

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

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

ate) 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 Specifying Connectors

The most interesting aspect of WRIGHT is its approach to specifying connectors. The roles of a connector describe the possible behaviors of each participant in the interaction, while the glue describes how these behaviors are combined to form a communication.

4.1 Notation

Our approach is to describe these behaviors as interacting protocols defined in a subset of CSP [6]. (In what follows, we will assume some familiarity with CSP.) The subset of CSP that we adopt is the use of events, and processes built out of primitives (e.g., STOP) and prefixing ($\rightarrow$), deterministic choice ($\Box$), and non-deterministic choice ($\sqcap$). We also allow names to be associated with a (possibly recursive) process expression.

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

$$P = (\textbf{let } Q = expr1 \textbf{ in } R)$$

Third, as in CSP, we allow events and processes to be labeled. The event e labeled with label l is written $l.e$. 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 $\surd$. (The reason for this will become clear later.) We use the symbol Σ to represent the set of all unlabeled events.

The subset of CSP that we have chosen makes the process descriptions "finite-state". Later we explain our rationale for this decision. However, most of the discussion that follows would carry forward without modification if we used a more complete subset of CSP.

4.2 Connector Description

To describe a connector type we provide process descriptions for each of its roles and its glue. As a simple example, consider the client-server connector, introduced informally in Section 4.2. Ignoring transmission of data, this is how it might be written using the notation just outlined.

```
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 2: A Pipe Connector

```
connectorService =
    role Client = request→ result → Client ⊓ √
    role Server = invoke→ return → Server □ √
    glue = Client.request→ Service.invoke→Service.return→
           Client.result→glue □ √
```

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 (□) the choice of invoke or √ is determined by the environment of that role (which, as we will see, is 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 (⊓) 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 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 — 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.

As more substantive example, Figure 2 illustrates the definition of an infinite pipe connector type. The complexity of this definition arises from the need to account for the possibility that either role may decide to stop. If the writer stops, the reader must be prepared to accept an "end of file" marker.

4.3 Connector Semantics

Intuitively, the roles of a connector act as independent processes constrained only by the glue, which orchestrates the interactions between the roles. Formally, the meaning of a connector is the parallel composition of its role and glue processes, where we arrange things so that the alphabets of the roles do not intersect and the alphabet of the glue includes the union of the events of the roles.

Definition 4.1 The *meaning of a connector description* with roles R_1, R_2, ..., R_n, and glue *Glue* is the process:

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

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

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

•

Here the glue's alphabet is 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 any of the processes to terminate successfully. Thus, successful termination of a connector is the joint responsibility of the all parties involved.

5 Connector Instantiation

Component ports are also specified by processes: The port process defines the expected behavior of the component at that point of interaction. For example, the request port of a client that makes a single request and then terminates successfully might look like:

component =
  port Request = request$\rightarrow$ result $\rightarrow$ $\surd$
  other ports...

To use a connector to define a particular system we must create an instance of the connector and then "attach" it by associating component ports with the connector roles. (See Figure 1.) But what does this mean?

At first glance it might seem that there should be no problem to solve. If the port protocols are identical to the role protocols, then we can simply substitute the ports for the roles in the overall system description.

But, in general, we would not like to require that the port and role protocols be identical. As a simple example, note that the above port protocol required only one request for service while the role allows an infinite number. Similarly, we can well imagine a client-server connector that allows two kinds of services to be performed, but that a particular port only requires the use of one. As

another example, a port that writes to a pipe may be designed to continue forever; in that case its protocol would not involve the *close* event.

Allowing that we would like to permit the role and port protocols to be different, we note that the port protocol defines the concrete interaction behavior of the component with its environment. Thus, when instantiated, the ports take the place of the roles in the actual system. It is reasonable, therefore, to define an instantiated connector as one in which all of its roles have been replaced by the ports of the components that it connects. Formally,

Definition 5.1 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:

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

•

But this now raises the key question: when is it legal to perform such an instantiation? We refer to this as the *port-role compatibility* problem.

As the examples above illustrate, it should be possible for roles not to exhibit all of the behavior allowed by a connector. On the other hand there are certain kinds of behavior that a port should not be allowed to exclude. For example, if a server must be initialized before a request is made, then the port had better include initialization as part of its promised behavior.

Evidentally what is needed is a definition that allows the port to ignore optional (i.e., nondeterministic choices), while respecting the obligations of the connector (i.e., deterministic choices). But this is precisely the notion of refinement!

Unfortunately it is not possible to use the notion of CSP refinement directly. There are two reasons for this. The first reason 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. (In this context, $P_{+B} = P \| STOP_B$). We extend the port's alphabet to that of the role, and vice versa.

The second, and more important, reason is that even if the port and role have the same alphabet it may be that the port process allows incompatible behavior, but that this behavior could 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, then the anomalous situation will not arise.

Thus to evaluate compatibility we need concern ourselves only with the behavior of the port *restricted to the contexts in which the role might find itself*. That is, to evaluate the suitability of a port to fill a given role, it is sufficient to consider the port's behavior over traces that are allowed by the role. 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 [2].)

We are led to the following definition of compatibility (where "\" is set difference) :

Definition 5.2 P **compat** R ("P is compatible with R")
if $R_{+(\alpha P \setminus \alpha R)} \sqsubseteq P_{+(\alpha R \setminus \alpha P)} \parallel det(R)$ •

Under these definitions, we see that port $Request$ = request → result → $\surd$ is compatible with role Client, but that it would not have been if the client had required an initialization event before a request.

6 Deadlock Freedom

An important goal in defining connector types is to be able to provide guarantees about the properties of their instances. If this were not possible there would be little benefit in having reusable connector types, since we would have to reestablish the properties of each connector instance whenever it is used.

One such property is deadlock-freedom. Intuitively, a deadlock-free connector is one in which the roles never get "stuck" in the middle of an interaction, each role expecting the others to take some action that can never happen. That is, if one of the connector's roles is prepared to make progress it should be possible for the connector as a whole to do so. On the other hand, we would like to allow the possibility that the connector as a whole can terminate successfully. For example, a client-server connector should allow the client to terminate the interaction, provided it does so at a point expected by the server. Similarly the pipe connector should allow termination when the writer closes the pipe.

In terms of our model of connectors, successful termination amounts to a joint transition (of all the roles and glue) to $\surd$, the process that announces success and then stops. (Recall that we have set up our renaming operator so that $\surd$ can be a shared event of all the roles and the glue). We can make this formal:

Definition 6.1 A connector C is *deadlock-free* if for all $(t, ref) \in failures(C)$ such that $ref = \alpha C$, then we have $tail(t) = \surd$. •

As argued above, such a property is only useful if it is preserved across connector instantiation. That is, we would like to be able to claim that an instance of a deadlock-free connector remains deadlock-free when instantiated by compatible ports.

Such a result would, of course, be trivially true if we used ports that were identical to the roles. But as we have argued above, ports and roles need not be identical. Less obvious, but equally true, is the fact that if ports are strict refinements of the roles then deadlock freedom is also preserved. This follows from the monotonic nature of process refinement, which requires the failures of a refinement to be a subset of the failures of the process it is refining. In other words, the refined process can't refuse to participate in an interaction if the role could not also have refused.

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

Consequently, it is not immediately clear whether deadlock-freedom is preserved across compatible port substitutions. In fact, it is not. The problem arises if the glue permits behaviors outside the range of those defined by the roles of the connector. Suppose, for example, that the glue allows a behavior of the form "$\ldots \Box R1 : crash \rightarrow \text{STOP}$" and that the event *crash* is not in the alphabet of role $R1$. Then the connector could be deadlock-free (in the sense defined above). Now consider a port that contains the same behavior (i.e.,$\ldots \Box R1 : crash \rightarrow \text{STOP}$). It is possible for this port to be compatible with role $R1$. But the connector can deadlock if the port is substituted for the role in that connector.

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

Definition 6.2 A connector $\text{C} = Glue \parallel (\text{R}_1{:}r_1 \parallel \text{R}_2{:}r_2 \parallel \ldots \parallel \text{R}_n{:}r_n)$ is *conservative* if $traces(Glue) \subseteq traces(\text{R}_1{:}r_1 \| \text{R}_2{:}r_2 \| \ldots \| \text{R}_n{:}r_n)$ •

Armed with this definition we can now state the desired result:

Theorem 6.3 *If a connector* $C = Glue \parallel (\text{R}_1{:}R_1 \parallel \text{R}_2{:}R_2 \| \ldots \| \text{R}_n{:}R_n)$ *is conservative and deadlock-free, and if for* $i \in \{1..n\}$, P_i **compat** R_i, *then* $C' = Glue \| (R_1{:}P_1 \| R_2{:}P_2 \| \ldots \| R_n{:}P_n)$ *is deadlock-free.*

7 Conclusion and Future Prospects

In this paper we have illustrated how refinement can be applied to the problem of specifying software architectures. This work is motivated by the need for a practical formal basis on which software engineers can develop architectural designs using common vocabularies of components, connectors, and patterns of composition.

As a step in that direction we have focused on reusable specifications of architectural connectors. Given such specifications, refinement emerges as natural way to understand when it is legal to use a connector in a given context. We have also illustrated that pure refinement is not sufficiently flexible and provided a somewhat more permissive definition that permits reuse of connectors in a larger number of contexts. We further claimed that this definition is not too loose: it is still possible to guarantee preservation of properties across connector instantiation.

The specific notation used here was a subset of CSP, embedded in the Wright architectural description language. That subset was deliberately chosen to produce finite state processes. As a result, all of the major properties described in the paper (deadlock-freedom, conservatism, and compatibility) can be checked automatically by tools such as FDR [3]. In fact, we are incorporating FDR into an architectural design environment, and plan to use it routinely to check the properties of connectors and their instantiations.

However, the finite nature of the examples should not mislead the reader into thinking that the results apply only in the case of finite state processes. Indeed, all of the results carry over to full CSP. Of course, automated checking is no longer possible in that case. We also believe that the approach outlined in this paper extends beyond CSP itself. In principle, it should be possible to use

alternative specification languages to define the protocols of connectors. For example, timed CSP could be used to express timing constraints on interactions to explain the behavior of such things as server timeouts. The notions of compatibility and conservatism would then have to be correspondingly augmented.

Acknowledgements

The results of this paper were developed jointly with Robert Allen. Several of the examples are adapted from other accounts of this work [1, 2]. The ideas have benefited substantially from comments by Steve Brookes, Daniel Jackson, Mary Shaw, and Jeannette Wing.

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.

References

[1] Allen, R. and Garlan, D. *Beyond Definition/Use: Architectural Interconnection.* in: **Proceedings of the ACM Interface Definition Language Workshop**. SIGPLAN Notices, Portland, OR, 1994.

[2] Allen, R. and Garlan, D. *Formal Connectors.* no. CMU-CS-192, Carnegie Mellon University, 1993. *In preparation.*

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

[4] Garlan, D. and Delisle, N. *Formal Specifications as Reusable Frameworks.* in: **VDM'90: VDM and Z – Formal Methods in Software Development**. Springer-Verlag, LNCS 428, Kiel, Germany, 1990, pp. 150–163.

[5] Garlan, D. and Shaw, M. *An Introduction to Software Architecture.* in: **Advances in Software Engineering and Knowledge Engineering, Volume I**, edited by V.Ambriola and G.Tortora. World Scientific Publishing Company, New Jersey, 1993.

[6] Hoare, C. **Communicating Sequential Processes**. Prentice Hall, 1985.

[7] **Open Systems Interconnection Handbook**. edited by G. R. McClain. Intertext Publications McGraw-Hill Book Company, New York, NY, 1991.

[8] Metalla, E. and Graham, M. H. *The Domain-Specific Software Architecture Program.* no. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.