The Wright Architectural Specification Language

Robert Allen David Garlan

Draft of 24 September 1996
CMU-CS-96-TBD

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

An important step towards establishing an engineering discipline of software is to provide a formal basis for describing and analyzing software architectures. The WRIGHT Architectural Specification Language was developed for this purpose. The key novel features of WRIGHT are its support for (a) formal specification of new architectural connection types, (b) formal definition of architectural styles, and (c) rules for checking the consistency and completeness of architectural designs.

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

Keywords: software architecture, architectural specification, architectural description languages, formal models

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 [SG96a].

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 such descriptions may provide useful documentation, the current level of informality limits their usefulness. It is generally not clear precisely what is meant by these 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.

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. Ideally we should be able to check that the overall description is consistent in the sense that the parts fit together appropriately. More ambitiously, we would like to be able to formally describe architectural families—or styles—and prove general properties about that family.

The WRIGHT Architectural Specification Language was developed for this purpose. WRIGHT provides a formal basis for specifying both the structure and behavior of architectural descriptions. Specifically, it supports the description of architectures as hierarchical graphs of components and connectors. Each component and connector is augmented with specifications that permit one to characterize precisely the abstract behavior of the components and their interactions. It also provides a set of rules for statically checking certain important properties of an architectural description, including (a) whether the expectations of component about its architectural environment are consistent with those actually provided by the other components in the architecture; (b) whether an architectural description is complete; and (c) whether the interface of a component is consistent with the computation performed by the component.

In addition to supporting formal specification and analysis of specific architectures, WRIGHT also supports the definition of *architectural styles*. An architectural style defines a family of systems that have a common architectural design vocabulary and satisfy a common set of design constraints. By permitting reasoning at the level of architectural styles, WRIGHT supports the powerful technique of developing general theories for classes of system architecture, thereby guaranteeing satisfaction of desirable properties for any specific architecture that satisfies the constraints of the style.

In this report we present the WRIGHT language. We begin by motivating its design. Next we provide an informal description of its key features for describing architectural structure. In the next two sections we explain how to model the behavior of an architecture. This is followed by an informal description of the validation checks that are supported by WRIGHT. We then present a formal semantics for WRIGHT and its associated analyses. Appendix A provides a BNF description for WRIGHT; Appendix B lists a complete example of a simple architecture defined in WRIGHT; Appendix C briefly discusses mechanical checking of WRIGHT specifications.

2 Language Goals

Developing a good architecture—one that satisfies the immediate product requirements, and eases future product evolution—is arguably one of the most critical aspects of practical software design. However, currently the ability to select or design appropriate architectures is impeded by the lack of good notations and tools for describing, analyzing, and manipulating architectures.

While this gap in technology is generally recognized, there is much less consensus about exactly what a formal notation for architecture should consist of, and what its detailed objectives should be. For example, should the language deal only with structure, or should it also support the description of other properties of interest? And, if the latter, which properties? It is important, therefore, to be clear about the intended goals of any proposed architectural formalism.

The goal of WRIGHT is to provide a *practical, semantically well-founded* language for characterizing the *abstract behavior* of architectures and architectural *families:*

...practical... real software architects should be able to use the notation to describe real systems. This has two consequences. First, the language must match engineers' intuitions and expectations about architectural description. In our view, it is not enough to have a language that only theoretians can use. Rather, it should supplement, not replace the now informal, but useful, concepts for describing architectures. Second, there should be tangible, incremental benefits for using the language. This means that the effort of specifying an architecture should be repaid in useful insight and analytic capability. Further, the engineer should be able to get some benefit even with small amounts of formalism.

...semantically well-founded... the language should be based on solid theoretic underpinnings that permit formal reasoning and analysis. While it is not necessary that engineers understand those theoretical underpinnings, it is critical that the derived capabilities for specification and analysis be sound.

...abstract behavior... we believe that a useful architectural specification language must go beyond simple structure. Therefore, WRIGHT focuses on certain kinds of architectural behavior: namely, the abstract computations and interactions that take place in a system. While there are many aspects of behavior that WRIGHT does not address (such as timing), it does provide a vehicle for characterizing the architecturally important events of a system and understanding how different components of the system cooperate to produce overall behavior. The choice of events, however, is up to the specifier: by picking a small number of high-level events of interest, a system can be modelled at a high level of abstraction, while a more detailed event model allows a more refined description and analysis.

...families... it should be possible to understand the properties of a collection of architectures that share some common features. By specifying the properties of a family of systems, the efforts of formalization can be amortized over a (usually infinite) set of potential systems in the family. Moreover, the ability to define families supports the specification and codification of established architectural styles that ease the design task of an architect.

We now show how WRIGHT attempts to achieve these goals. As we will illustrate, the key features that contribute are: (a) the use of architectural elements (components and connectors) to structure specifications, (b) support for defining and analyzing new architectural connection types, (c) semantic foundations based on process algebra, (d) analytic capability for determining (statically) whether a system is well-formed and complete, and (e) the ability to define architectural styles.

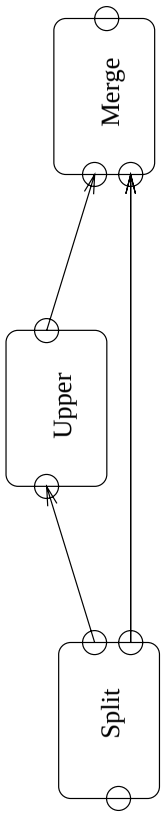

Figure 1: Box-and-line drawing of example system.

3 The Structure of WRIGHT

WRIGHT is built around the basic architectural abstractions of *components*, *connectors*, and *configurations*. WRIGHT provides explicit notations for each of these elements, formalizing the general notions of components as loci of computation, connectors as patterns of interaction, and configurations as (possibly hierarchical) graphs of components and connectors. In this section we describe these basic structural notations and show how they are combined to describe the software architecture of a system. To emphasize the structural aspects of WRIGHT, we defer a discussion of the notations for specifying computation and interaction until Section 4.[1]

Throughout this section, we will use the simple example architecture illustrated in Figure 1 to explain the basic ideas. This architecture is a pipe-filter system that reads a stream of characters from its input, outputting the same stream of characters, but with every other character capitalized. To accomplish this, three components are used: Split, Upper, and Merge. In Split, the input stream is divided into two streams. The first stream is sent through Upper, which capitalizes every character, and the second is left unchanged. The two streams are then recombined by Merge. Characters are transmitted from one component to another via pipe connectors, which provide asynchronous, infinite-capacity, in-order transfer of data.

[1] In addition to simplifying the presentation, deferring the details of behavior specification also underscores the fact that the structural framework of WRIGHT can be viewed independently from the specific notation used for behavior specification. If there are critical properties not captured by WRIGHT's behavior specifications, the structuring provided by WRIGHT is still useful.

Component SplitFilter
  Port Input ⟨*read data until end-of-data is reached*⟩
  Port Left ⟨*output data repeatedly*⟩
  Port Right ⟨*output data repeatedly*⟩
  Computation ⟨*repeatedly read from* Input*, then output, alternating between* Left *and* Right *ports.*⟩

Figure 2: The structure of a component description

3.1 Components

A component is a locus of computation. As in the example system above, a typical component might read characters from its input and convert each letter to upper case. A database system might include a repository component that provides access to its data and a client component that summarizes the data in a report when it is requested by the user.

In WRIGHT, each component is defined by a component type description, which has two important parts: an *interface* and a *computation*. The interface consists of a number of *ports*. Each port represents an interaction in which the component may participate.[2] For example, Split in Figure 1 might be defined by the component type SplitFilter, shown in Figure 2. The description indicates that each component of this type has three ports, one for input and two for output. As an alternative example, a map database server component might have two ports, one to respond to clients' queries about the map, and another that an administrator would use to update the map.

The computation part of a component describes what the component actually does. The computation carries out the interactions described by the ports and shows how they are tied together to form a coherent whole. In the above example, the behavior of a SplitFilter component is to divide its input into two streams.

A port specification describes two aspects of a components interface. First, it partially describes the component's intended behavior. Specifically, the port specification defines the behavior of the component as viewed through the lens of that particular port. The port becomes, in effect, a *partial specification* of the component.

The second aspect of a port is a specification of the expectations about the system within which the component will interact. In the example, SplitFilter illustrates this aspect of a port by indicating (shown informally above) that it *expects* to be able to read data on Input until it is notified of end-of-data.

In SplitFilter, notice how each port specification says something about the **Computation**. The Input port indicates how data is read, and the Left and Right ports say something about the output behavior. Note, however, that the port specifications do not in combination completely define the full behavior of the computation. In particular, they are unable to relate the behaviors at different ports. For example, there is no indication outside the **Computation** that Left and Right alternate. The **Computation** is itself the full specification upon which analysis of the component's properties will be based. However, the ports act as a useful partial specification of a component's behavior, which, as we will see later, allows us to simplify the analysis of an assembled system.

Thus ports are analogous to interface specifications in traditional module-oriented languages. In both cases the specifications provide an *abstraction* of the component's actual behavior, and permit

[2]Here, and throughout the report, we will often use the term "component" instead of the term "component type". It will be clear from context when we are referring to a type and when to an instance.

certain kinds of checks that uses of the component are consistent with its definition. However, there are three fundamental differences. First, in WRIGHT a component can have multiple ports—and hence multiple interfaces. Second, as we have noted, the port not only declares what it provides to the system, but also declares what expects of its environment. Third, as we will see later, port specifications provide more than static "signatures" of the interface—they also indicate dynamic patterns of interaction.

3.2 Connectors

In WRIGHT, connectors define patterns of interaction between components. One of the novel features of WRIGHT is the ability to make these patterns explicit as new connector types.

Explicit Connector Types

In the informal diagrams of today's architectural descriptions, the lines—or connectors—that join components often represent rich and varied abstractions of component interaction. For example, a line representing a pipe connector might indicate sequential flow of data between two filters. Or it might represent a procedure call connector that uses a call-and-return pattern of control. An event-multicast connector might represent an abstraction in which one component can announce events that are received by an arbitrary set of listener components. More complex connectors include such things as database connectors (e.g., supporting two-phase commit) and reliable, secure network message-passing connectors.

To support the need of architects to represent many kinds of interaction, WRIGHT allows one to define new connector types. Once a new type of connector is defined, a system can make use of any number of instances of that connector. By capturing a pattern of interaction as a connector type, and then using the pattern repeatedly in connector instances, WRIGHT makes explicit the commonality that typically occurs throughout a software architecture. For example, the "pipe" interaction is used multiple times in the simple pipe-filter example above. If each pair of components were to specify the interaction independently (as required by more traditional module-oriented languages), then there would be no simple means to verify that there is indeed only one kind of interaction taking place throughout the system.

The identification of commonality is important because it allows us to reuse general results about the shared element. For example, if we can show that the pipe connector *type* has the property that no data is lost, then that property will hold wherever any pipe *instance* is used in the system. If we had to do the analysis in the context of a specific interaction, (for example, between the Left output of Split and the Input of UpperCase) then would have to repeat the analysis each time we used the pipe connector.

The explicit description of connector types is also valuable in the larger context of a development process as well. By specifying explicitly that each use of the connector instance uses an identical interaction pattern we can exploit shared infrastructure to implement the connector correctly. For instance, if we use a runtime library to implement a communication pathway (such as `stdio.h` to implement pipes), how can we tell where the library can be used to connect components? If the library corresponds to a connector type, the architecture makes the answer clear. If there is no such correspondence, a potentially difficult verification task looms. ahead

The use of explicit connectors also increases the independence of components by structuring the way a component interacts with the rest of the system. A connector provides, in effect, a set of requirements must be met by components using the connector, and an information-hiding boundary that clarifies what expectations the component can have about its environment. Thus

Connector Pipe
  Role Source ⟨*deliver data repeatedly, signalling termination by closing the pipe*⟩
  Role Sink ⟨*read data repeatedly, closing at or before end of data*⟩
  Glue ⟨*deliver data in order from* Source *to* Sink⟩

Figure 3: The structure of a connector description

a component specification is described in a way that allows it to be used in multiple contexts. In the Split filter, for example, the Left port refers not to the UpperCase filter, which is the target of data in our example system, but to a general interaction pattern. The SplitFilter specification does not indicate whether the output is delivered to UpperCase, some other filter, several other filters, a file, or even if that output is dropped from the system and ignored. The component specification need only indicate what that component will do, because the connector specifications are there to describe how the component is combined with others in an actual context of use.

Connector Structure

A WRIGHT description of a connector consists of a set of connector *roles* and the connector *glue*. Each role specifies the behavior of a participant in the interaction. For example, a pipe has two roles, the source of data and the sink (the component that receives the data). A procedure call connector has a caller and a definer. An event broadcast connector has an announcer and zero or more listeners.

Connector roles indicate what is expected of the components that will participate in that interaction. In Figure 3, for example, the Sink role indicates how that participant is expected to behave: Any component that acts as a Sink is permitted to read data and is responsible for closing the connection. This role might, for example, be filled by the Input port of the SplitFilter. The SplitFilter does indeed read data and does not continue beyond end-of-data.

The connector **Glue** describes how the participants work together to create an interaction. In the case of a pipe (shown in Figure 3), **Glue** describes how the data from the source is delivered to Sink. As another example, a procedure call connector glue would indicate that the caller role initiates an invocation, followed by a return from the callee role.

As with the **Computation** of a component, the **Glue** specification of the connector defines the full behavioral specification. As we will see when we define the semantics of a WRIGHT configuration, the **Glue** processes coordinate the components' behavior. In effect, we interpret a connector specification to mean that *if* the actual components obey the behaviors indicated by the roles, *then* the computations of the components will be combined as indicated by the **Glue**.

3.3 Configurations

In order to describe a complete system architecture, a set of components and connectors of a WRIGHT description must be combined into a *configuration*.

Instances

To use the component and connector types in a specific system an architect must first define and name a set of instances of those types. Instance declarations for the example of Figure 1 are shown

```
Configuration Capitalize
    Component UpperCase
        ...
    Connector Pipe
        ...
    ...
  Instances
    Split : SplitFilter
    Upper : UpperCase
    Merge : MergeFilter
    P1, P2, P3 : Pipe
  Attachments
    Split.Left as P1.Source
    Upper.Input as P1.Sink
    Split.Right as P2.Source
    Merge.Right as P2.Sink
    Upper.Output as P3.Source
    Merge.Left as P3.Sink
end Capitalize.
```

Figure 4: The structure of a configuration.

in Figure 4.

Attachments

Once component and connector instances have been declared, a configuration is created by describing a set of *attachments*. Attachments define the topology of the configuration by indicating which components participate in which interactions. This is done by associating a component's port with a connector's role.

For example in Figure 4, the attachment declaration "Split.Left **as** P1.Source" indicates that the component Split will play the role of Source in the interaction P1. It will fill this role through the port Left. That is, all of the data that Split outputs to port Left will be delivered to whichever component is the sink of pipe P1. In Figure 4, the matching declaration "Upper.Input **as** P1.Sink" indicates that it is the component Upper that will receive the data from Split.

The attachment declarations bring together all of the elements of an architectural description. Each component carries out a computation, part of which is determines the interactions specified by its ports. Those ports are attached to a roles, which indicate what rules the ports must follow in order to be a legal participants in the interactions specified by the connectors. If each component, as represented by its respective ports, obeys the rules imposed by the roles, then the connector **Glue** defines how the **Computations** are combined to form a single, larger computation for the system as a whole.

Hierarchy

WRIGHT permits hierarchical descriptions. In particularl, the computation for component (or the

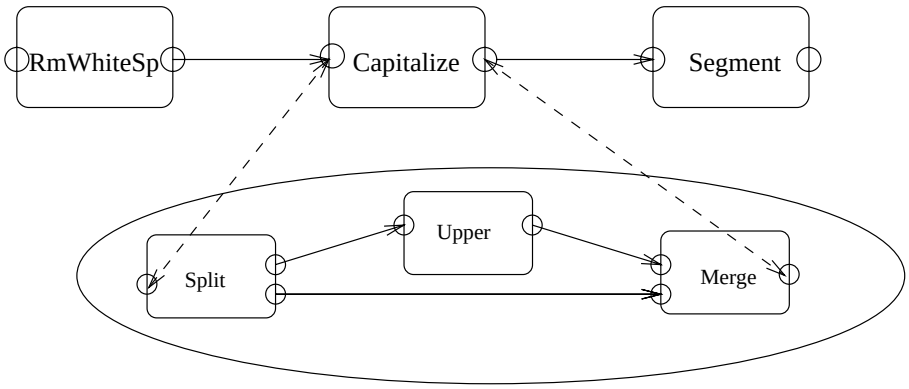

Figure 5: Hierarchical Architecture.

glue for a connector) can be represented either by a primitive behavior specification (to be described in Section 4) or by an architectural description itself. In the later case, the component serves as abstraction boundary for an architectural subsystem.

When a component is represented by an architectural subsystem, it is described as a configuration in the same way as indicated above. In addition, however, for a component the nested architectural description has an associated port map, which defines how the port names on the "inside" are associated with the port names at interface of the component. (Similarly for roles: role names on the "inside" are identified with role names on the "outside".)

Figures **??** and 6 and illustrate the use of hierarchy on a simple example. The system is responsible for removing white space from a stream of characters, capitalizing every other letter, and then segmenting the output into 5-character words separated by spaces. This is accomplished at the top level with three filters: the first removes white space, the second capitalizes every other character, and the third produces 5-character words . The middle component, however, is realized as the subarchitecture outlined earlier.

3.4 Style

So far we have discussed how the architecture of an individual system is described in WRIGHT: the architect introduces component and connector types, creates instances of those types, and then attaches them to form a configuration.

In many situations, however, an architect is actually concerned with a *family* of systems, rather than a single one. In WRIGHT such families are termed *architectural styles*.

The Importance of Architectural Styles

The interest in architectural styles (or families) arises for several reasons. First, the architect might be interested in developing a framework for some product line. By specifying the shared architectural structure of the products, the architect can define standards to which the individual products must adhere. This in turn simplifies the development of a specific system in the product line, makes it easier to understand and maintain those systems, supports reuse of shared implementations and infrastructure, and often allows an organization to develop domain-specific, reusable componentry and component generators.

Second, even when a system is not part of a product line, it will likely undergo various revisions over time. That sequence of releases itself determines a family. Thus when developing the architec-

```
Configuration Cap-Words
    Component RmWhiteSp
        ...
    Component CapitalizeFilter
        Port Input ...
        Port Output ...
        Computation
            Configuration Capitalize
                Component UpperCase
                ...
            Instances
                Split : SplitFilter
                Upper : UpperCase
                ...
            Attachments
                Split.Left as P1.Source
                Upper.Input as P1.Sink
                ...
            end Capitalize
            Bindings
                Split.input = Input
                Merge.output = Output
            end Bindings
    Component Segment
        ...
    Connector Pipe
        ...
    ...
  Instances
    ...
  Attachments
    ...
end Cap-Words
```

Figure 6: Hierarchical Specification in WRIGHT.

Style Pipe-Filter
  Connector Pipe
    Role Source ⟨*deliver data repeatedly, signalling termination by close*⟩
    Role Sink ⟨*read data repeatedly, closing at or before end of data*⟩
    Glue ⟨Sink *will receive data in same order delivered by* Source⟩
  Interface Type DataInput = ⟨*read data repeatedly, closing the port at or before end-of-data*⟩
  Interface Type DataOutput = ⟨*write data repeatedly, closing the port to signal end-of-data*⟩
  Constraints
    $\forall\, c : connectors \bullet type(c) =$ Pipe
    $\forall\, c : components; p : Port \mid p \in ports(c) \bullet type(p) =$ DataInput $\vee\ type(p) =$ DataOutput
End Style

Configuration EveryOther
    Style PipeFilter
    ...
End Configuration

Figure 7: The structure of a style.

ture of a system, an architect typically also considers the dimensions of evolution permitted over time for that product.

Third, some architectural styles are constrained in such a way that they can guarantee certain desirable properties. By picking a style with the right properties, the architect can ensure that the specific system under design will inherit those properties.

For example, if an architect is faced with a problem of transforming a sequential stream of input in a regular way, the pipe-filter style of systems permits the architect to decompose the transformation into a collection of simpler transformations. By using the pipe-filter style, the architect knows that the simple transformations can be combined easily, without worrying about declaring complicated interfaces or control sequences. If the system has performance constraints, there are known techniques for analyzing the critical computation path and for distributing a pipe-filter system across several processors.

If, on the other hand, the system is faced with the task of maintaining a complex data store, and with keeping it up-to-date in a potentially unreliable environment, the architect will instead look for a database-transaction style such as X/Open [GR93]. This style of system construction permits the architect to rely on the data integrity properties (e.g., "ACID" properties) guaranteed by the style. Other styles range from the domain-independent, layered system style to domain-specific styles for systems such as robotic controllers and employee payroll systems. (For a more detailed discussion about architectural styles and their benefits see [SG96b, GAO94, AAG95].)

Specifying Styles in WRIGHT

A style defines a set of properties that characterize a family of software architectures. These properties include the use of a common vocabulary of architectural elements and restrictions on the ways that vocabulary can be used to create architectural configurations. Figure 7 illustrates the form of a WRIGHT style definition. (Appendix B contains the full description.)

WRIGHT provides three facilities for defining a style: (a) component, connector, and interface types; (b) parameterization; and (c) constraints.

3.4.1 Component, Connector, and Interface Types.

As with configurations (see Figure 4) a style can introduce a common architectural vocabulary by declaring a set of component and connector types. As illustrated in Figure 7, the pipe-filter style provides Pipe connector type. When a configuration is defined using the pipe-filter style, Pipes are automatically available for use.

In addition to complete component and connector types, a style may define certain aspects of a component or connector. For example, in the pipe-filter style all components are filters, which use only dataflow for input and output. This commonality of filters needs to be exposed, although the filter computations will differ between different filters. Also, the names and numbers of input and output ports will differ from filter to filter. For example, in the earlier example SplitFilter has one input and two outputs, while the UpperCase filter has one of each.

To define shared interface properties, a WRIGHT description can introduce *interface* types. For example, the pipe-filter style introduces DataInput and DataOutput interface types for defining the roles of a pipe and for later use in defining filters. The details of interface types will become clearer in Section 4, when the notations for behavior are introduced, and in Section 7.8, when the tests for attaching a port to a role are discussed.

Parameterization

To permit additional flexibility in defining new component, connector and interface types WRIGHT, supports *parameterizing* of type descriptions. Informally, this capability permits a description of a type to leave "holes" in the description that will be filled in when the type is instantiated for a specific configuration.

For example, in the Unix pipe-filter style (a specialization of the example style that we have been considering), all components have one input, named Stdin, and two outputs, named Stdout and Stderr. The interface to all Unix filters is the same. But the computation performed by each Unix filter is different. Thus to describe a general Unix filter type, we must leave a hole in the description so that the computation can be specified for the particular filter. We can describe the Unix filter as a parameterized component type, as follows:

Component Unix-filter(C : **Computation**)
  Port Stdin = DataInput
  Port Stdout = DataOutput
  Port Stderr = DataOutput
  Computation = C

We can then use this description to describe any number of Unix filters:

Upper : Unix-filter(⟨*pass output, translating to uppercase*⟩)
Lower : Unix-filter(⟨*pass output, translating to lowercase*⟩)
LaTeX : Unix-filter(⟨*translate input in .tex form to .dvi form; error messages are sent to* Stderr⟩)

WRIGHT permits any part of the description of a type (including its ports, roles, computation, glue, interface names) to be replaced with a *placeholder*, which is then filled with a parameter when the type is instantiated. In addition, as we will see when we discuss formal behavior descriptions, *part* the behavior of a port, role, computation, or glue can also use placeholders.

In addition to leaving simple holes in type descriptions, another way of parameterizing a type description is by *number*. Suppose, for example, that a particular class of filter system uses many filters that split their input among a number of outputs. SplitFilter, in Figure 2, is a good example. But in our earlier description it has exactly two outputs. Suppose we want three, or four? For a style to explicitly list all of the possible Split filters with any number of outputs, we would need an infinitely long description. Instead, we make the number of outputs a parameter to the description:

Component SplitFilter(nout : 1..)
  Port Input = DataInput
  Port $\text{Output}_{1..\text{nout}}$ = DataOutput
  Computation = ⟨*read from* Input *repeatedly, writing to* Output_1, Output_2, etc. *in succession*⟩

A parameter accepting a range of integers is written "⟨*lowerbound*⟩..⟨*upperbound*⟩." ⟨*lowerbound*⟩ indicates the smallest acceptable integer, and ⟨*upperbound*⟩ indicates the largest. If one of the bounds is omitted (as is the case with ⟨*upperbound*⟩ in the SplitFilter), this indicates that there is no limit in that direction. Thus, the SplitFilter can accept *any* positive integer as its parameter.

In the body of the WRIGHT description, the number parameter can be used to control the number of particular kinds of ports or roles that can appear. A port or role description that can have multiple copies is indicated by specifying a range of integers as a subscript to its name. In the SplitFilter example, there can be more than one Output port, depending on the value of the nout parameter. It is not permitted to omit either bound in the use of a range on a port or role name. (We can't have an infinite number of ports, after all.)

Thus, the number of ports on a component is determined at instantiation time and cannot be changed during the execution of the system. This reflects the *static* nature of a WRIGHT description. WRIGHT assumes that, at run time, the set of components and the interaction topology do not change. Many kinds of "dynamic" architectures (where components appear and disappear, or where the network of communications changes during an execution) can be modelled in one of two simple ways. First, one may include all *potential* elements in the system description, and then ignore those that do not currently exist. Second, one may describe each possible configuration as a different architecture—in effect, use a mini-style to describe a single system. In each case, the number of ports, roles, components, and connectors will be finite at any point in time.

Constraints

Interface types and parameterization facilities could equally well be used in the definition of a single configuration. If a system is large enough to contain repeated types, or if the architect wants to emphasize commonalities between different parts of a single architecture, these elements of the language can be useful even if the system does not refer to a separate style definition.

But a style is more than just a vocabulary that may be used to define configurations. It isn't enough for the pipe-filter style to have pipes and data inputs and outputs *available* to be used. In order for a system to be in the pipe-filter style, it must use *only* these elements. In a more restrictive pipeline style, even this may not be enough. The components must be strung together by pipes in a single line.

When an architect constructs a system as a member of a larger family of systems, exploiting special properties of that family, or modifies an existing system that has been validated based on a specific set of assumptions, the architect will wish to refer to an explicit statement of what constraints apply. For example, if a system happens to contain only pipes and filters, is this coincidence because no other constructs turned out to be necessary, or is it an intrinsic property of

the style that is being relied on for specific purposes (such as simplifying the implementation basis for the system).

To specify these kinds of constraints, a WRIGHT style description may declare properties that must be obeyed by any configuration in the style. For example, the pipe-filter style would indicate that all connectors must be pipes:

$$\forall\, c : \mathit{connectors} \bullet \mathit{type}(c) = \mathsf{Pipe}$$

In addition, the style would require that all components in the system use only DataInput and DataOutput ports:

$$\forall\, c : \mathit{components}; p : \mathit{Port} \mid p \in \mathit{ports}(c) \bullet \mathit{type}(p) = \mathsf{DataInput} \vee \mathit{type}(p) = \mathsf{DataOutput}$$

Each of the constraints declared by a style represents a predicate that must be satisfied by any configuration declared to be a member of the style. The notation for constraints is based on first order predicate logic. The constraints refer to the following sets and operators:

- *components*: the set of components in the configuration.
- *connectors*: the set of connectors in the configuration.
- *attachments*: the set of attachments in the configuration. Each attachment is represented as a pair of pairs ((*comp*, *port*), (*conn*, *role*)).
- *Name(e)*: the name of element e, where e is a component, connector, port, or role.
- *Type(e)*: the type of element e.
- *Ports(c)*: the set of ports on component c.
- *Computation(c)*: the computation of component c.
- *Roles(c)*: the set of roles of connector c.
- *Glue(c)*: the glue of connector c.

In addition, any type that has been declared as part of the style's vocabulary may be referred to by name. As we saw in the examples above, the pipe-filter style introduces Pipe, DataInput, and DataOutput, and the constraints of the style refer to these types by name.

Here is a more complex example of a constraint. It indicates that a configuration must have a "star" topology.

$$\begin{array}{l} \exists\, \mathit{center} : \mathit{components} \bullet \\ \quad \forall\, c : \mathit{connectors} \bullet \exists\, r : \mathit{Role}; p : \mathit{Port} \mid ((\mathit{center}, p), (c, r)) \in \mathit{attachments} \\ \forall\, c : \mathit{components} \bullet \exists\, cn : \mathit{connectors}; r : \mathit{Role}; p : \mathit{Port} \mid ((c, p), (cn, r)) \in \mathit{attachments} \end{array}$$

The first predicate indicates that there must be a component "center" that participates in every interaction of the system. The second predicate indicates that every component must participate in some interaction, thus guaranteeing that every component is connected to "center".

4 Specifying Behavior

We have illustrated how to specify the structure of an architecture, and how an architectural style can be used to describe a family of similar structures. But thus far our specifications of the meaning of the architectural types and of the behavior of the elements has been informal. What precisely does it mean to say that the pipe ensures that "Sink will receive data in the same order delivered

by Source?" How does a component signal end of data on a DataOutput port? How can we specify which participant is in control at any point in an interaction and what are the restrictions on the order in which things happen?

The behavior and coordination of components is specified in WRIGHT using a notation based on CSP [Hoa85]. CSP is a notation for specifying patterns of behavior and interaction. We will begin by provided a brief, informal introduction to CSP. (The interested reader should consult a text on CSP for more detailed explanations.)

Events

The basic unit of a CSP behavior specification is an *event*. An event represents an important moment or action. For example, end-of-data is an event for the Sink of a pipe. Similarly, the write event represents the delivery of data by the Source. The same event can occur many times in a complete behavior. For example, the source of a pipe can write data many times.

Because we are interested in how different components control interactions, we add notation to CSP to distinguish between *initiating* an event and *observing* an event. An event that is initiated by a process is written with an overbar: The specification of the DataOutput port would use the event $\overline{\mathsf{write}}$ to indicate that it initiates this event. The DataInput port, on the other hand, observes end of data, so in its specification this event would be written without an overbar: end-of-data. We also refer to initiated events as *signalled* events: The pipe mechanism, for example, signals end of data, so its event would be written $\overline{\mathsf{end\text{-}of\text{-}data}}$.

A special event in WRIGHT is $\surd$, which indicates the successful termination of a computation. Because this event is not actually a communication event, it is not considered either to be initiated or observed. When we refer to an event but don't care whether it is initiated or observed, we will say that a process *engages* in the event. Thus, processes indicate correct termination by engaging in $\surd$.

An important property of events is that they can carry data. If a process supplies data, this is considered output, and written with an exclamation point. For example, the source of data for a pipe supplies data when it writes to the pipe using $\overline{\mathsf{write!x}}$. If a process receives data, this is input, and written with a question mark: e?x. Notice that output is usually signalled ($\overline{\mathsf{e!x}}$) and input is usually observed (e?x).[3]

Processes

In CSP patterns of event behavior are (for historical reasons) called *processes*. Processes are described by combining events and other, simpler processes. The most primitive process is **STOP**, the process that does nothing.

The simplest way of constructing a new process is *sequencing*. Given a process P and an event e, the process e→P is the process that first engages in the event e and then behaves as P. For example, we define the *success process*, §, to be $\surd$→**STOP**, the process that successfully terminates immediately.

Another form of sequencing is the ";" operator. This combines two processes in sequence. P ; Q is the process that behaves as P until P terminates successfully and then behaves as Q. For example (e→f→§) ; (g→§) = e→f→g→§.[4] If the process P does not terminate, then P ; Q acts as P forever.

[3] This is not always the case, however. Consider a component that reads data from a variable. While the component initiates this event, it inputs data. Similarly, the glue of the variable connector represents the delivery of the value as an observed output event (*i.e.* an output event that it does not initiate).

[4] We use the convention that → associates to the right: e→f→P = e→(f→P).

If sequencing were the only operator, it would not be possible to describe very interesting behaviors. The only processes would be those that engage in a single string of events, of a fixed length, and then stop. In order to describe more complex behaviors, we need *naming*, *state*, and *alternatives*.

By naming processes, we can describe behavior patterns that occur repeatedly. For example, consider the following process definition:

$$\mathsf{P} = \mathsf{e}{\rightarrow}\mathsf{P}$$

The process named P performs the event e and then acts as the process P. This is a *recursive* definition. The overall behavior of this process is to do e over and over again, without ever stopping. Named processes can also be introduced into other processes using **where**:

$$\mathsf{f}{\rightarrow}\mathsf{P} \textbf{ where } \mathsf{P} = \mathsf{e}{\rightarrow}\mathsf{P}$$

This process does a single f and then repeats e over and over.

We can add state to a process definition by adding subscripts to the name of a process: P_i is a process with a single state variable, i. For example,

$$\mathsf{P}_1 \textbf{ where } \mathsf{P}_i = \overline{\mathsf{count}!i}{\rightarrow}\mathsf{P}_{i+1}$$

is a process that counts: $\overline{\mathsf{count}!1}$, $\overline{\mathsf{count}!2}$, $\overline{\mathsf{count}!3}$, *etc.*

Sometimes, however, we want a process to have different behavior depending on the value of its state variables. For example, we might want a circular counter, that counts to three and then resets: 1, 2, 3, 1, 2, ... A state dependency is introduced with a *conditional* definition, written by adding a test on the state variables:

$$\begin{array}{l}\mathsf{P}_V = \mathrm{Q} \\ \quad p(V)\end{array}$$

defines a process P over variables V only when the boolean expression $p(V)$ is true. For example:

$$\begin{array}{rl}\mathsf{P}_1 \textbf{ where } \mathsf{P}_i = & \overline{\mathsf{count}!i}{\rightarrow}\mathsf{P}_{i+1} \\ & i < 3 \\ \mathsf{P}_3 = & \overline{\mathsf{count}!3}{\rightarrow}\mathsf{P}_1\end{array}$$

defines the circular counter.

Another important way of extending the behavior of a process is through alternatives. The first kind of alternative is a process that recognizes the possibility of two behaviors in its environment. This is termed *deterministic* or *external choice*. It uses the operator $\Box$. The process $\mathsf{e}{\rightarrow}\mathsf{P} \,\Box\, \mathsf{f}{\rightarrow}\mathrm{Q}$ is the process that will behave as the process P if it first observes the event e and will behave as the process Q if it first observes the event f. This is called deterministic choice because the behavior of the process is entirely determined by what the environment does. Deterministic choice is typically made between observed events.

The second kind of alternative is a process that makes an internal choice about which of two behaviors to perform. We call this *non-deterministic* or *internal choice* and use the operator $\sqcap$. The process $\overline{\mathsf{e}}{\rightarrow}\mathsf{P} \sqcap \overline{\mathsf{f}}{\rightarrow}\mathrm{Q}$ is the process that will either initiate $\overline{\mathsf{e}}$ and then act as P or initiate $\overline{\mathsf{f}}$ and then act as Q.[5] The process itself decides which to do, without consulting the environment. Thus, non-deterministic choice is typically made between initiated events.

[5]Both alternative operators bind more closely than sequencing. Thus, $\mathsf{e}{\rightarrow}\mathsf{P} \,\Box\, \mathsf{f}{\rightarrow}\mathrm{Q} = (\mathsf{e}{\rightarrow}\mathsf{P}) \,\Box\, (\mathsf{f}{\rightarrow}\mathrm{Q})$.

To make processes more flexible, the sequencing and choice operators can also be quantified over a set: $\forall\, x : S\ \langle op \rangle \mathsf{P}(x)$. This operator constructs a new process based on a process expression and the set S, combining its parts by the operator $\langle op \rangle$. For example,

$$\forall\, i : \{1,2,3\} \Box\ P_i = P_1 \Box P_2 \Box P_3.$$

If the sequencing operator ';' is used the result is some unspecified sequencing of the processes:

$$\forall\, x : S\ ;\ P(x) = \forall\, x : S \sqcap (P(x)\ ;\ \forall\, y : S \setminus \{x\}\ ;\ P(y)).$$

Thus, $\forall\, i : \{1,2,3\}\ ;\ \mathsf{P}_i = (\mathsf{P}_1; \mathsf{P}_2; \mathsf{P}_3) \sqcap (\mathsf{P}_1; \mathsf{P}_3; \mathsf{P}_2) \sqcap (\mathsf{P}_2; \mathsf{P}_3; \mathsf{P}_1) \sqcap (\mathsf{P}_2; \mathsf{P}_1; \mathsf{P}_3) \sqcap (\mathsf{P}_3; \mathsf{P}_1; \mathsf{P}_2) \sqcap (\mathsf{P}_3; \mathsf{P}_2; \mathsf{P}_1)$.

Examples

Given these notations, we can now specify the behavior of architectural elements precisely. As a simple example, consider the basic procedure call connector.

The basic idea of a procedure call is that there is one party, the Caller that initiates the procedure call invocation. The other party, the Definer, carries out the defined computation, and then returns. This pairing can be carried out multiple times.

A formal WRIGHT specification of this interaction (ignoring data) is as follows:

Connector Procedure-call
  Role Caller = $\overline{\mathsf{call}} \rightarrow \mathsf{return} \rightarrow \mathsf{Caller} \sqcap \S$
  Role Definer = $\mathsf{call} \rightarrow \overline{\mathsf{return}} \rightarrow \mathsf{Definer} \Box \S$
  Glue = $\mathsf{Caller.call} \rightarrow \overline{\mathsf{Definer.call}} \rightarrow$ **Glue**
    $\Box\ \mathsf{Definer.return} \rightarrow \overline{\mathsf{Caller.return}} \rightarrow$ **Glue**
    $\Box\ \S$

There are three elements of this definition worth noting. First, the Caller and the Definer use different alternative (or choice) operators to indicate their different roles. The Caller decides whether to initiate a procedure call or not, and so it uses the non-deterministic choice operator. The Definer, on the other hand, offers the option of a procedure call, so it uses deterministic choice. It is up to the other parties (in this case the Caller) to determine whether a call or termination will occur.

Second, because the **Glue** mediates the interaction between multiple participants, its specification must indicate *which role's* event is indicated in any situation. This is done by prefixing each event by the name of a role. So Caller.call indicates that the Caller component executes the call, and Definer.call indicates the Definer component being notified of the call.[6]

Third, the **Glue** indicates how the behavior of the roles *combines* to form a complete interaction. Each of the two main branches of the **Glue** process indicate how an event of one participant triggers another event in the other participant. Because the **Glue** does not take the point of view of any one of the roles, its use of initiated and observed events is complementary to that of the roles: If a role initiates an event, it is observed in the **Glue**. If a role is to observe an event, it must be initiated by the **Glue**.

Thus, "$\mathsf{Caller.call} \rightarrow \overline{\mathsf{Definer.call}} \rightarrow$ **Glue**" indicates that the Definer will observe a call event following its initiation by the Caller. "$\mathsf{Definer.return} \rightarrow \overline{\mathsf{Caller.return}} \rightarrow$ **Glue**" indicates that the Caller will process a return following the signal by Definer.

[6] In practice, these are actually a single occurrence in the software system, but for technical reasons WRIGHT treats events in different roles to be distinguishable events.

This particular glue structure, where an event that is initiated by one role (thus, observed by the glue) is always echoed at another role, is quite common in connector interactions. In fact, it is so common that many architecture description languages do not permit any other form of glue. We do not wish to restrict the kinds of interaction patterns that can be described in WRIGHT to just this simple class, but we can provide "syntactic sugaring" to simplify glue descriptions for common cases.

As a more interesting example, consider how we might complete the formalization of the pipe-filter style. The earlier specification of a DataOutput port was informal: ⟨*write data repeatedly, closing the port to signal end-of-data*⟩. This can be formalized as follows:

Interface Type DataOutput = $(\overline{\text{write!x}} \rightarrow \text{DataOutput}) \sqcap (\overline{\text{close}} \rightarrow \S)$

A DataOutput port has two events with which it communicates, write and close. Both of these events are initiated by the component, and so they are written with an overbar. The component decides between the two events, as indicated by the ⊓ operator. If the component chooses to write, it provides a data element (indicated by the !x), and then makes the choice again (it behaves as DataOutput). If the component chooses to close, then it must terminate without writing again (§ is the only option following close).

Formalizing the DataInput port is slightly more complicated. Informally, we said ⟨*read data repeatedly, closing the port at or before end-of-data*⟩. This seems to indicate a similar kind of choice between reading and closing. But when we attempt to read, there might not be any data available (end-of-data may have been reached). In this case, we *must* close. All of these situations are covered in the formal definition:

Interface Type DataInput = $(\overline{\text{read}} \rightarrow (\text{data?x} \rightarrow \text{DataInput} \;\Box\; \text{end-of-data} \rightarrow \overline{\text{close}} \rightarrow \S)) \sqcap (\overline{\text{close}} \rightarrow \S)$

In order to fully define the interaction represented by the Pipe connector, we need to specify three behaviors: The Source role, the Sink role, and the **Glue**. Not surprisingly, since they were designed to go together, the Source and Sink role definitions use the DataOutput and DataInput definitions respectively. The **Glue**, which indicates how the behaviors of the two sides are combined to form a complete interaction, is more complicated. It must constrain the read and write events so that the Sink receives exactly the data that the Source has produced. Furthermore, the **Glue** must coordinate the closing of the pipe by the Source with signalling end-of-data to the Sink. The following definition accomplishes this:

$$
\begin{aligned}
&\textbf{Glue} = \text{Open}_{\langle\rangle} \\
&\quad \textbf{where } \text{Open}_{\langle\rangle} = \text{Source.write?x} \rightarrow \text{Open}_{\langle x \rangle} \\
&\qquad\qquad \Box\, \text{Source.close} \rightarrow \text{Closed}_{\langle\rangle} \\
&\qquad\qquad \Box\, \text{Sink.close} \rightarrow \text{Capped} \\
&\quad \text{Open}_{s\langle x \rangle} = \text{Sink.read} \rightarrow \overline{\text{Sink.data!x}} \rightarrow \text{Open}_{s} \\
&\qquad\qquad\quad \Box\, \text{Source.write!y} \rightarrow \text{Open}_{s\langle x \rangle\langle y \rangle} \\
&\qquad\qquad\quad \Box\, \text{Source.close} \rightarrow \text{Closed}_{s\langle x \rangle} \\
&\qquad\qquad\quad \Box\, \text{Sink.close} \rightarrow \text{Capped} \\
&\quad \text{Closed}_{s\langle x \rangle} = \text{Sink.read} \rightarrow \overline{\text{Sink.data!x}} \rightarrow \text{Closed}_{s} \\
&\qquad\qquad\quad \Box\, \text{Sink.close} \rightarrow \S \\
&\quad \text{Closed}_{\langle\rangle} = \text{Sink.read} \rightarrow \overline{\text{Sink.end-of-data}} \rightarrow \text{Sink.close} \rightarrow \S \\
&\qquad\qquad \Box\, \text{Sink.close} \rightarrow \S \\
&\quad \text{Capped} = \text{Source.write!x} \rightarrow \text{Capped} \\
&\qquad\qquad \Box\, \text{Source.close} \rightarrow \S
\end{aligned}
$$

This definition recognizes three main situations in the interaction: both ends of the pipe are open (represented by the process Open), the Source has closed (represented by Closed), or the Sink has closed (represented by Capped). Once both parties have closed, no further interaction is possible, so the Glue becomes §. While there is a possibility of data flow (*i.e.* the Sink has not closed), the data available is kept as state associated with the process (either Open or Closed). A write event by the Source adds to the queue of data, and a read event by the Sink removes from it. Once the Source has closed, and there is no more data available, the Glue will be in the state $\text{Closed}_{\langle\rangle}$. In this situation, the Sink will be informed of end-of-data, and then only close is possible.

Notice how in this connector, the **Glue** does more than simply match up events in the various roles. Anywhere a pipe is used in an architecture, there is an additional buffer that is added to the system, containing the data that has been written by the source, but not yet read by the sink. At the same time, the connector retains its abstract task of showing how the computations of the various participants are combined to form a larger computation: The purpose of the pipe connector is to allow the architect to *ignore* this buffering of data and concentrate on finding or constructing the filters necessary to achieve the desired data transformations.

The SplitFilter can also be completed using formal behavior specifications and the pipe-filter style:

Component SplitFilter(nout: 1..)
  Port Input = DataInput
  Port $\text{Output}_{1..nout}$ = DataOutput
  Computation = Transfer_1
    where $\text{Transfer}_i = \overline{\text{Input.read}} \rightarrow (\text{Input.data?x} \rightarrow \overline{\text{Output}_i\text{.write!x}} \rightarrow \text{Transfer}_{i+1}$
      $\square$ Input.end-of-data $\rightarrow$ Close)
    $1 \leq i \leq nout$
    $\text{Transfer}_{nout+1} = \text{Transfer}_1$
    $\text{Close} = \overline{\text{Input.close}} \rightarrow \forall\, i : 1..nout \;;\; \overline{\text{Output}_i\text{.close}} \rightarrow \S$

5 The Behavior of WRIGHT Configurations

Now that we can describe the structure of an architecture and assign behaviors to each of the elements, we need to address two important questions: (a) What do the behavior patterns imply about the system as a whole? and, (b) How can we decide if a description is valid? This section addresses these questions by showing how the descriptions are combined and by giving rules that must be obeyed by legal WRIGHT descriptions. This section is intended to give enough information about these issues for the practical user of WRIGHT. Technical details appear in Section 7.

As we saw in the previous section, behaviors are specified by combining events into patterns called processes. There is a process for each of the elements of a WRIGHT description, one for each port, role, computation and connector glue. Of these, the port and role specifications represent the *interfaces* to the components and connectors, while the computation and glue represent the overall, actual behavior of the components and connectors, respectively. In this section we explain how these processes work together to define the behavior of the configuration and how they can help us to determine whether the configuration contains inconsistencies.

5.1 Parallel Composition

Abstractly, we think of the behavior of an architectural configuration as consisting of the behaviors of the individual components, each operating independently except that they are coordinated by

the glue of the connectors to which they are attached. That is, the computation of each component forms a part of the overall behavior, where the order in which the computations occur and the transfer of data from one to the other is coordinated by the connectors.

The basic technique used in CSP to model the combination of coordinated processes is *parallel composition*. Two processes are composed in parallel (indicated in CSP with the operator ||) by having both processes control which events can occur. If both processes agree on an event then the event can occur. For example, consider two processes, P and Q:

$P = (e \rightarrow f \rightarrow P) \,\Box\, (g \rightarrow P)$, and
$Q = e \rightarrow (f \rightarrow Q \,\Box\, h \rightarrow Q)$.

What will happen if we combine these in parallel, as P||Q? At first, P permits either e or g. But Q may only engage in e, so this is what will happen. Once e has occurred, Q may now engage in either f or h. But now P is only capable of f, so f occurs. After the $\langle e, f \rangle$ sequence, both P and Q are in their original states, so the sequence repeats. Thus, the process P||Q is equivalent to the process $R = e \rightarrow f \rightarrow R$.[7]

5.2 Alphabets

In most systems each component is involved in only a small subset of the total set of events of the system. In these situations a process that does not make any reference to a system event should not interfere with other components that do engage in that event For example, if we have a system with two filters in a pipeline (see Figure 8), then the first component (labelled A) should not have any control over the computation to be performed by the second (labelled B). A's only influence on B should be via its use of the connector, C. So we can't simply put the computations in parallel and give each computation full control over the events in the system. Let's look at a simple example to clarify this problem. Consider three CSP processes, A, B, and C, and think of them as the two components and connector in Figure 8.[8]

$A = a \rightarrow A \sqcap \S$
$C = a \rightarrow c \rightarrow C \,\Box\, \S$
$B = c \rightarrow b \rightarrow B \,\Box\, \S$

The intent of this system is as follows: the component A will engage in the event a some number of times and then decide to terminate. The component B is capable of executing the event b any number of times, or terminating. B will execute b whenever it observes c. The connector C is responsible for ensuring that whenever a occurs, c follows. Thus, for each a, the connector transmits a c, and this triggers a b event in B. The overall effect should be that there will be exactly one b for each a. This models a kind of connector where one component triggers a particular behavior in another component; for example, in a component that reacts to the receipt of a message from another component.

But what if we combine these processes in parallel: A||C||B? If every process controls every event, then nothing can happen! Initially, process A wants to do the event a. This is fine with C, but B wants to do the event c first. So a can't happen first. But neither can c, because neither A nor C agrees to this.

[7] In CSP, two processes are equivalent when they represent exactly the same pattern of behavior. Details of equivalence and its more flexible cousin, refinement, appear in Section 7.1.

[8] Because this is a simple example to show the properties of CSP, and not really a full WRIGHT example, we are ignoring the initiate/observe distinction for events.

Figure 8: A and B interact *only* via C.

To address this issue, CSP uses the concept of an *alphabet*. The alphabet of a process (written αP) is the set of events over which it has an influence. In a parallel composition, a process controls those events in its alphabet and ignores the others. In our example, the alphabet of A is the set {a}, the alphabet of B is {b, c}, and the alphabet of C is {a, c}. Thus, in the composition A‖C‖B, the event a *can* occur first, because B doesn't have it in its alphabet. Both A and C react to it occurring, but B ignores it. Once a has occurred, then B and C can agree on c, and A ignores it. After that, either a or b can occur, since none of the processes will prevent them. A and C agree on a, and B can do b on its own.

The problem of process alphabets in CSP is quite similar to the problem of scoping often encountered in system design. When we consider a particular aspect of a software system, a procedure call or the state of some variable, say, how do we determine how much of the system we have to look at in order to cover all of the influences on the part we are interested in? In some programming languages there is no easy way to avoid looking at all of the code in the system. If we use techniques such as data abstraction or have a software architecture, then there are clear scoping boundaries that we can use to limit the potential effect on each other.

5.3 Application to WRIGHT

Let us now apply the notion of parallel composition to WRIGHT specifications.

We want to set up the formal interpretation so that a component's **Computation** interacts only according to the constraints of the connector **Glue** to which it is attached. That is, each computation should proceed independently of the other components, except that the events published in the interface (the ports) should be coordinated via the glue processes of the attached connectors, as we would expect from Section 3.

Basically, we combine the behavior specifications of each instance of an architectural element in the system via parallel composition, as we discussed above. That is, there will be a process for each component instance and one for each connector instance. But there are two difficulties:

1. Behavior specifications are associated with a type, not an instance: How can we combine multiple uses of a type in single system?

2. The types' specifications are *context-independent*: How can the attachment declarations be used to ensure that the right interactions take place? If we look at the way behaviors are specified in a component's computation and a connector's glue, none of the event names match up. The glue will refer to an event with a role name, and the computation will refer to it by a port name. Recall from the pipe-filter example (Section 4) that in the pipe **Glue**, the events describing the output of a filter are Source.write and Source.close. But in a **Computation**, the events describing the output of one particular filter, say the SplitFilter, are Output_i.write and Output_i.close. If we just combine these using ‖, they won't match up.

Both of these are problems that are common whenever types are used in software engineering, and whenever any attempt is made to reuse elements, either within a single system or across

multiple systems. In object-oriented definitions, for example, one must distinguish between *class* and *instance* variables, depending on whether the value is to be shared by object in a class or there is a separate variable for each individual object. By the same token, if we wish to replace one runtime library with another, either the two libraries must be forced to use exactly the same set of procedure names, or the program must be changed in order to incorporate the new library. These solutions are often costly, if not impossible. By using local names in the WRIGHT specification these problems are avoided.

WRIGHT's local event names are converted into CSP's global events using *renaming functions*. A renaming function takes a process and changes all of the names of its events. For example, consider a function *shift* that shifts each event by one letter, a to b, b to c, *etc.* When *shift* is applied to a process, $P = a \rightarrow b \rightarrow P$, the result is a process with the same structure, but with different event names: $shift(P) = b \rightarrow c \rightarrow shift(P)$.

We use two different kinds of renamings to combine the types' behavior specifications into an overall behavior of a WRIGHT configuration. The first is used to make multiple copies of the specifications for instances. These functions add the names of the instances to each event name, and are called *labelling functions*. Thus, an instance of the SplitFilter named Splitter would refer to its events with the name Splitter. So we would get Splitter.Left.write, Splitter.Left.close, *etc.* Thus multiple instances of a type will not interfere.

The second kind of renaming matches up the names of attached ports and role. If we have an attachment declaration,

Splitter.Left **as** P_1.Source

for example, these functions make sure that all of the events for the Left port in the Splitter's computation match up with the events from the Source role of the P_1 glue. Thus Splitter.Left.close would be the same event as P_1.Source.close after the attachment renaming functions are applied.

To see how does this works out consider a simple example. Suppose we want to make the three processes A, B, and C from above into actual WRIGHT components and connectors. They would look something like this:

Configuration ABC

Component A-type

Port Out = $\overline{a} \rightarrow$ Out $\sqcap$ §

Computation = $\overline{\text{Out.a}} \rightarrow$ **Computation** $\sqcap$ §

Component B-type

Port In = c$\rightarrow$In $\Box$ §

Computation = In.c$\rightarrow\overline{b}\rightarrow$**Computation** $\Box$ §

Connector C-type

Role Origin = $\overline{a} \rightarrow$ Origin $\sqcap$ §

Role Target = c$\rightarrow$Target $\Box$ §

Glue = Origin.a$\rightarrow\overline{\text{Target.c}}\rightarrow$**Glue** $\Box$ §

Instances

A:A-type

B:B-type

C:C-type

Attachments
A.Out **as** C.Origin
B.In **as** C.Target **end** ABC.

The resulting CSP process is the following:[9]

$$
\begin{aligned}
&A = \overline{A.Out.a} \rightarrow A \sqcap \S \\
\| \; &C = A.Out.a \rightarrow \overline{B.In.c} \rightarrow C \; \Box \; \S \\
\| \; &B = B.In.c \rightarrow \overline{B.b} \rightarrow B \; \Box \; \S
\end{aligned}
$$

6 Validating Descriptions

We have shown how WRIGHT can be used to express the structure and behavior of a software architecture, as well as to describe the constraints on a family of systems that form an architectural style. Specifically, we have seen how WRIGHT allows us to describe connectors explicitly, distinguishing different connectors with different protocols over events, clarifying what parties are responsible for initiating those events, and showing what parts of an interaction are visible to each participant. We have also seen that WRIGHT permits the computation of each component to be described in terms of significant communication events, and divides the interface of each component into several distinct interactions. Finally, the parameterization and style constraint facilities permit individual descriptions to be generalized to a pattern of interaction or computation, and to place more global restrictions on how systems can be assembled.

As a formal specification language, however, WRIGHT has value beyond enabling architects to write down an architectural description. Another important aspect of the language is its support for analysis and reasoning about the system thus described.

There are many kinds of analysis that one might consider at the architectural level of design, including analyses include functional correctness of the system, potential for expansion to meet increasing demands, contention for critical resources, and probable cost to implement subsystems. Each of these analyses rests on different properties of the described system, and would be suitably supported by different architectural formalisms.

Two criteria for an architectural description that underly all of these analyses, however, are consistency and completeness. Informally, consistency means that the description makes sense; that different parts of the description do not contradict each other. Completeness is the property that a description contains enough information to perform an analysis; that the description does not omit details necessary to show a certain fact or to make a guarantee. Thus, completeness is *with respect to* a particular analysis or property.

Consistency and completeness are fundamental to architectural analysis because without them, none of the other analyses makes sense. Consistency is necessary before the description can be said to describe an actual system. If one part of an architectural description indicates one thing, and another part the opposite, surely one of them is wrong.

Because the architectural level of design is fundamentally focussed on questions of structuring and composition, consistency among parts is especially critical at this level of design. The principles that ensure that a system will function as a coherent entity must be built in as part of the overall

[9] In fact, the CSP interpretation removes the initiate/observe markings on events so that they will synchronize. They have been left in here to make the connection to the WRIGHT description clearer.

1. Port-Computation Consistency (component)
2. Connector Deadlock-free (connector)
3. Roles Deadlock-free (role)
4. Single Initiator (connector)
5. Initiator Commits (any process)
6. Parameter Substitution (instance)
7. Range Check (instance)
8. Port-Role Compatibility (attachment)
9. Style Constraints (configuration)
10. Attachment Completeness (configuration)

Figure 9: Summary of Tests

structure; if the abstract description is inconsistent, any refinement or implementation of it will likely retain that inconsistency.

Completeness is important to an architectural description because an analysis can only be based on what we actually know about a system. If an architect analyses the communication behavior of a component, but the system description leaves out part of the interface, what good is the analysis? For example, an analysis of contention for a resource can only be applied if *all* of the parts that access that resource are known.

The problem of completeness is especially critical for the architect because of the importance of *abstraction* at this level of design. There is always a tension between the need to include critical information that is necessary to guarantee important system properties and the risk of cluttering the architecture with constraints and details that can make the architecture unwieldy and difficult to work with.

In the next sections, we discuss the questions of the consistency and completeness of a software architecture instance description. For each of these properties, we show how WRIGHT addresses these issues and structures analysis of an architectural description to highlight any inconsistency or incompleteness, and how simple tests can be applied to guarantee that an architectural description is both consistent and complete. The discussion that follows is not completely formal; the main point is to discuss the kinds of criteria that must be met and to show how WRIGHT approaches these problems. Formal details of all definitions and checks can be found in Section 7.

For reference, a complete list of the consistency and completeness tests is shown in Figure 9, along with an indication of to what each test applies in parentheses.

6.1 Consistency

In the description of an architectural instance, the property of consistency is important for each element of an architectural description: the components, the connectors, and the configurations. For each of these elements we must ask ourselves how we can tell if a description is consistent.

Components

When we say that a description is internally consistent, we mean that none of the parts of the description contradicts any of the other parts. That is, whenever two parts of the description overlap, describing the same property or behavior, the two parts must agree. If one part of the description describes something not covered by another part, there is no possibility of contradiction.

As we have seen, for a component description the parts consist of an interface, decomposed into a number of ports, each of which describes an interaction in which the component participates, and a computation, which describes the full internal behavior of the component. To determine the consistency of a component description, we must determine whether the computation violates the constraints of the interface. That is, *to the extent that a computation is involved in carrying out a particular interaction*, the computation must obey the rules of interaction defined by the port.

Recall that we do *not* view the collection of ports as representing a full specification of the component, which must then be verified. The purpose of the ports is to ensure that the component's behavior in an interaction meets the requirements of the interaction, and therefore we only care that the computation is *consistent* with the ports: If the computation is consistent with the port, and the port is compatible with the role, then the computation is compatible with the connector.

Also recall that a port specification indicates two aspects of a component. First, it indicates some aspect of the component's behavior, and second, it indicates the expectations of a component about the system with which it interacts.

In WRIGHT we can compare the behavior of the component with the behavior of a port by checking whether the port represents a *projection* of the component's overall behavior. A port is a projection of the component if the component acts in the same way as the port when we ignore all events not in the port's alphabet. For example, consider the following specification:

Component Double
  port In = read?x →In □ close→§
  port Out = $\overline{\text{write!x}}$ →Out ⊓ $\overline{\text{close}}$→§
  Computation = In.read?x →$\overline{\text{Out.write!(2*x)}}$ →**Computation** □ In.close→$\overline{\text{Out.close}}$→§

In this specification, the port In is a projection of the **Computation** over the events that are prefixed with the name In (*i.e.* In.read and In.close). Out is a projection of the **Computation** over the events that are prefixed with the name Out (*i.e.* Out.write and Out.close). When we say we ignore a given set of events, we really mean that we hide them from the external environment, treating them as internal choices: If we hide In.close and In.read, then the deterministic choice between them in the **Computation** becomes an internal, or non-deterministic choice between the visible events that follow them: $\overline{\text{Out.write}}$→**Computation** ⊓ $\overline{\text{Out.close}}$→**Computation**. This, of course, matches the port Out.

The second aspect of a port, indicating expectations about the environment, is illustrated by the following example. Suppose we attempt to attach Double's In port to an interaction that might initiate events other than read and close? For example, a role might state:

role FailingIn = read?x→FailingIn □ close→§ □ fail→§

This specification indicates that a component filling this role might also be informed of a failure, after which it would be expected to stop using the port. The **Computation** of Double doesn't handle the fail event, and the system would break if Double found itself in a situation where the fail event occurred.

Fortunately, the interface of Double (port In) clarifies the situation. It indicates that Double may *not* be used in the interaction represented by FailingIn, because the port specification shows that Double expects either In.read or In.close, while the role indicates that In.fail should also be handled. So, as we will see later in the section on attachments, FailingIn does not meet the expectations of Double, and can not be attached to the In port: the specification of Double does not have to deal with the fail event.

So if a port specification does not refer to an event, the component is not required to handle it. But there are situations in which it might be appropriate to have a component specification describe behaviors that it does not expect to occur. One such situation occurs with *generalized computation* or *reuse*. It might be simpler to describe a computation by including all cases, not just those that are covered by the interface. Or, we might want to reuse a more general computation specification. For either of these reasons, the port specification might only cover a subset of the situations that the component can actually handle.

For example, suppose we have a server that calculates square-roots over all integers—including negatives. If we want to use it in a situation where complex answers are not acceptable, we would need to change the interface to indicate that only positive integers may be delivered by the client. It doesn't make sense to have to change the algorithm used internally by the server just to match the new, more restrictive interface.

Another reason a computation description might cover situations that don't appear in the interface is *bullet-proofing*. We might want to describe the behavior of the component Double in the face of failure, even though the interface specifies that the system must avoid it. This might be useful, for example, if there is a risk that the implementation of another component is broken. Here is an extended specification of Double:

Component Double
port In = read?x →In □ close→§
port Out = $\overline{\text{write!x}}$ →Out ⊓ $\overline{\text{close}}$→§
Computation = In.read?x →$\overline{\text{Out.write!(2*x)}}$ →**Computation** □ In.close→$\overline{\text{Out.close}}$→§ □ In.fail→§

In this specification, Double promises that if it observes a fail event on port In, then it will terminate immediately, and not attempt to use In or send any more output to Out.

But wait! Now the **Computation** process doesn't project into the Out port as we expect. Consider the computation ignoring In events, which we will name CompOut:

CompOut = $\overline{\text{Out.write!(2*x)}}$ →CompOut ⊓ $\overline{\text{Out.close}}$→§ ⊓ §

The non-deterministic choice of § means that Double may terminate without sending a close event on Out. The port specification Out, on the other hand, indicates that termination *must* be signaled with close.

But of course this isn't really a fair test. The whole point of leaving fail out of the In port specification was to indicate that the component doesn't have to worry about it happening. That is, the simple projection CompOut can only terminate without closing if the environment violates the assumption of port In that In.fail events will not occur. Therefore we must use a modification of the projection test to determine port/computation consistency:

Test 1 (Port/Computation Consistency) *A port specification must be a projection of the* **Computation**, *under the assumption that all other port interfaces are obeyed by the environment.*

We say that an interface is obeyed if the system supplies only those sequences of events which are covered by the port process. The technical details of this test (as for all of the tests described in this part) are given in Section 7.

Connectors

A connector represents a potential interaction among components. While the **Glue** indicates how the participants will be coordinated, the roles describe how the participants are expected to behave. Thus, the connector description must ensure that the coordination of the **Glue** is consistent with the expected behavior of the components, as indicated by the roles. We must ensure that the participants in the interaction will not become disastrously out of synch so that they are no longer truly communicating.

Suppose that there are two components in a system, a client and a server. Suppose also that the server provides a necessary value for the computation of the client, but it must receive an initialization signal before it can begin computation. The client, unaware of this, immediately requests the value from the server. Inevitably, this will result in disaster for the system: The server might return a bogus value, compromising the client's calculations, it might crash, disrupting other computations in progress, or it might simply ignore the client, leaving it stranded waiting for a return value. In any case, it is clear that we cannot say that the client and server are communicating in any meaningful way.

Because WRIGHT is based on CSP, this kind of error, where participants in an interaction cannot agree on the next appropriate event, can be detected as *deadlock*. A CSP process is said to deadlock when it may refuse to participate in all events, but has not yet terminated successfully (by participating in the $\surd$ event). Conversely, a process is deadlock-free if it can never get into a deadlock situation.

The situation described between the client and server would be detected in the WRIGHT situation as deadlock in the overall interaction between the client and server components. Because the server is waiting for an initialization event, while the client is only willing to provide a value request deadlock will result. While this could be detected in the behavior of the overall WRIGHT configuration, using the **Computation** processes of two actual component instances together with the **Glue** of the client-server connector, we can use the structure of WRIGHT to detect it using only the information of the connector type specification. This is because the roles contain enough information to detect the problem: If we use the roles as stand-ins for the components, (*i.e.* put them in parallel with the **Glue**) then they will deadlock, just as the components will. If they do not, but the components do, then there must be either an incompatibility between the roles and the ports on the components (indicating that the restrictions of the roles are not met by the component), or an inconsistency between the component's port and its computation, which is detected by test 1. Thus, inconsistencies between the participants in an interaction and the coordination of the **Glue** are detected by the following rule:

Test 2 (Connector Deadlock-free) *The* **Glue** *of a connector interacting with the role processes must be deadlock-free.*

Another kind of inconsistency is also detectable as deadlock: if a role specification is *internally* inconsistent. In a complicated role specification, there may be errors that lead to a situation in which no event is possible for that participant, even if the **Glue** were willing to take any event. This is avoided by another test:

Test 3 (Roles Deadlock-free) *Each of the roles in a connector must be deadlock-free.*

Notice that while this additional check is necessary for roles, the fact that the **Glue** is involved in every event in an interaction means that no separate rule is necessary for it: If the **Glue** deadlocks

then the composition of the **Glue** with its roles will deadlock, and therefore test 2 is sufficient to check internal **Glue** consistency.

Another kind of inconsistency can arise when we consider *control* of the interaction. What if all of the participants agree on what should happen next, but they can't agree on which component should do it? This kind of problem might occur, for example, if two components must communicate a value.

Suppose the developers agree that they will use a procedure call to pass the value between two components. Everything is fine, right? But what if they both choose to *declare* a procedure that can be used to communicate the value? One of the components declares a procedure that will deliver the value as a result, expecting the other party to "pull" the value, and the other declares a procedure that will accept a new value as a parameter, expecting the first component to "push" the value. The system will compile just fine, but when the system is executed, neither procedure will ever be called, because they will both wait for the other component to initiate the action. A similar problem occurs if both components assume the other component will declare a procedure and so they attempt to invoke a procedure. In this case, there will be no procedure available to call. The full system will not even link properly (although each individual component will compile).

While our example here is detected in the implementation as the common definition-use problem, this kind of conflict about initialization is not limited to cases in which one party must define an interaction and the other uses it. If there is a more complicated interaction specified by the connector (such as an event broadcast system or a networking protocol), it may be the case that neither party defines the interface, and they must determine which of them uses which part of the mechanisms available. For example, if two components use an event mechanism to send a message, and neither ever registers a callback the same problem as two declared procedures will arise, but the linker will be unable to detect the problem.

To detect these conflicts, it must be possible to distinguish between a component that controls an activity and one that simply observes, or reacts to, it. WRIGHT makes a distinction between initiated and observed events precisely to avoid this problem. Recall that in Section 4, we introduced an annotation on events to distinguish initiated events (with an overbar) from observed events (without an overbar). Initiated events are intended to represent those events where the described process takes some action, such as sending a message or invoking a procedure. Observed events represent situations where the process expects some other party in the environment to take action, and expects to react to the action (*e.g.* they might receive a message from some other party, or be invoked as a procedure).

A given event only makes sense, avoiding control conflicts, if there is exactly one process for which the event represents an action, and all other processes are observers of the event. Thus, there must be a *single initiator* of every event:

Test 4 (Single Initiator) *For every event in a connector type specification, exactly one of the roles or the* **Glue** *must initiate the event. All other processes must either observe it or omit it from their alphabet.*[10]

A final rule for connectors ensures that the initiate and observe notations on events are used consistently: we require that the *initiator commits* to an event. Consider the following extract from a possible WRIGHT specification:

Role Invalid = $\bar{e} \rightarrow P \;\square\; \bar{f} \rightarrow Q$....

[10] Of course, if one of the participants is the initiator of the event, then it must not also observe the event, or there could still be situations where no component is the initiator of the event.

In this specification the fact that e and f are initiated indicates that Invalid is the cause of these events. The component filling this role is responsible for making sure that they occur. But the operator $\Box$ indicates that the *environment* decides *which* of them will occur. How can this be? How can the environment control whether they happen while the component ensures that they do happen? In our view, this doesn't make sense, and the "initiator commits" rule ensures that such specifications do not arise.

Test 5 (Initiator Commits) *If a process initiates an event, then whenever it does so, it must commit to that single event without influence by the environment. By committing to the event, the process ensures that refusal by the environment to accept this event must result in a potential deadlock.*[11]

Configurations

Now that we have a notion of consistency for component and connector types, how can we determine if an architectural configuration is consistent? Essentially, we must determine whether the instance declarations and attachments combine to use the type declarations in meaningful ways.

Instance Declarations Since the only information added to a type by an instance declaration is the name of the instance and the value of the actual parameters, the question of consistent instance declaration boils down to two questions: (a) Is the name of the instance unique? and, (b) Have we supplied reasonable actual parameters? If a type declaration leaves a placeholder in its definition, then the instance declaration that supplies the element to be filled in must result in a reasonable declaration. For syntactic placeholders (such as role/port names) the only possible conflicts are the introduction of naming clashes (we don't want a connector to have two different roles with the same name, for example). For process placeholders (such as parameters that stand in for the **Computation** declaration, for example), conflicts can arise because the definition supplied with the instance declaration conflict with the fixed part of the type definition, or because two different parameters result in a conflict. These can be checked by substituting the actual parameters for their placeholders and then applying the type-based checks described above.

Test 6 (Parameter Substitution) *An instance declaration of a parameterized type must result in a valid non-parameterized type when the actual parameters are substituted for the formal parameters.*

In the case of numeric parameters, there is a special obligation for the type definer. Since there may be limits placed on the values that are permitted (by the range declarations) we require the type to guarantee that these limits are adequate to simplify the previous test to the following:

Test 7 (Range Check) *A numeric parameter must be no smaller than the lower bound, if declared, and no larger than the upper bound, if declared.*

What this means is that a type parameterized by number represents a *family* of types, one type for each possible parameter value. Every member of the family must be a consistent type. The reason that we can make this requirement for numeric parameters, but not, in general, for other parameters, is that there is no simple *syntactic* way of restricting the values that may be supplied for these other parameters.

[11] Technically, the process must refuse all other events. Note that this rule applies to component processes as well.

Attachments As we discussed above, 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 we also want to be able to attach a port that is not identical to the role.

The reason for this is that ports specify the interaction patterns of a single, concrete communication, and therefore are quite specific about what will occur, while roles specify the constraints on a general, abstract set of communications, and therefore specify a range of interactions. Consider, for example, the output interaction port of the component Double described earlier:

port Out = $\overline{\text{write!x}} \rightarrow$Out $\sqcap$ $\overline{\text{close}} \rightarrow \S$

This interface indicates that the component will provide data via the write event some unspecified number of times, and then close. It might be used to feed another part of the system in an interaction that covers exactly that situation:

Role Source = $\overline{\text{write!x}} \rightarrow$Source $\sqcap$ $\overline{\text{close}} \rightarrow \S$

So far so good. But what if we want to use another component, Gen3, instead of Double?

Component Gen3 =
Port Output = $\overline{\text{write!1}} \rightarrow \overline{\text{write!2}} \rightarrow \overline{\text{write!3}} \rightarrow \overline{\text{close}} \rightarrow \S$
Computation = $\overline{\text{write!1}} \rightarrow \overline{\text{write!2}} \rightarrow \overline{\text{write!3}} \rightarrow \overline{\text{close}} \rightarrow \S$

The port Output and the role Source are not an exact match (*i.e.* they are not the same process). But we should be able to use Gen3 in this interaction because it *does* supply data using write some (now specified) number of times and then close. So we can't use a rule of simple substitution.

On the other hand, we do need make sure that the port fulfills its obligations to the interaction. For example, we wouldn't want to use a component as Source if it didn't ever send the close event. For example the port

Port BadOut = $\overline{\text{write!x}} \rightarrow$BadOut $\sqcap$ $\S$

should not be acceptable.

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:[12]

Test 8 (Compatibility) *When in a situation described by the role protocol, the port must always continue the protocol in a way that the role could have.*

This means that the port must handle all of the observed events that the role specifies, but may possibly handle more, and that when choosing among events to initiate, the port must select from the set specified by the role, but may disallow options permitted by the role.

[12]This test is probably more clearly understood in its formal statement. See the discussion of Compatibility in Section 7.

Satisfying Style Constraints

So far, we have discussed tests that apply to WRIGHT descriptions independent of what style is selected for the architectural description, or even if the configuration has no declared style. As we noted in Section 3.4, however, a style description can impose additional constraints on any instance descriptions that are declared to be in that style. Recall that these are described as predicates over the instances.

A system configuration is consistent with its declared style if it obeys each of these constraints. That is, the style constraints represent a proof obligation on the architect, who must show that they are true for the described system:

Test 9 (Style Constraints) *The predicates for a style must be true for a configuration declared to be in that style.*

There is, of course, a much more outrageous form of inconsistency that can arise from the style constraints: If the constraints contradict each other or describe only illegal systems, there will not be any legal instances of the style! For example, what if we have two constraints that contradict each other:

$$\forall\, c : Components; p : ports(c) \bullet type(p) = DataOutput$$
$$\exists\, c : Components; p : ports(c) \bullet type(p) = DataInput$$

The first predicate states that every component has only DataOutput ports, and the second predicate states that at least one component has a DataInput port. Since no port can be both, there can not be any system that obeys both these constraints: The style is inconsistent.

6.2 Completeness

As we discussed in the introduction to this section, completeness is considered to hold with respect to particular properties or analyses. While a description may be complete with respect to one analysis, such as freedom from deadlock, it may be incomplete along another dimension, such as timing or datatype consistency.

WRIGHT handles completeness in different ways, depending on what property is being considered. For some properties, such as communication dependencies between components, WRIGHT builds completeness into the semantics of the language: All communications dependencies in an architecture are covered in a WRIGHT description *by definition*. If a dependency is omitted, then it does not exist in the system.[13]

For other properties, an incomplete description appears as an inconsistency in the architecture: If a glue description omits state values that are necessary to guarantee freedom from deadlock, then the connector, as described, will have the potential to deadlock (which is defined as an inconsistency in test 2 above). In the Pipe connector, for example, if the glue did not keep track of whether the Source role had signalled a close event, a deadlock could arise when the Sink role requests a data value that can never be delivered. The Sink will block forever, since the glue does not not have the information to deliver an end-of-data signal. This kind of completeness check differs from the completeness that is built into the language, because a check is necessary, but it is covered by the tests already described.

[13] Of course, there can still be an inconsistency between the architectural description and the implementation of the system as it is built. In this case, we would say that the implementation and the architecture describe *different* systems; practically, it may simply be that the architecture has omitted a detail.

With respect to some properties, WRIGHT is inherently incomplete. The CSP formalism on which WRIGHT is based does not contain any information about the timing of events, for example. Because there is no information about this in the description, the formalism must necessarily be considered incomplete with respect to any analyses that depend on it. If this kind of information is critical to an application, then it could be added as annotations to the WRIGHT structure, but WRIGHT as it exists does not include this information.

Another category of completeness with which we need to concern ourselves is intermediate between properties that are completely covered by the consistency checks and those that cannot be described in WRIGHT at all. These are properties of completeness that can be detected using the WRIGHT formalism but that are not covered by the consistency checks.

One important kind of completeness that falls into this category is the completeness of a configuration. When we describe a collection of components and connectors, how do we determine if the described system is capable of functioning without the addition of more parts? It may be that while all of the instances and attachments that *are* present are consistent, there is a critical attachment that has *not* been made. In this case, a component might depend on observing events that will never occur, or an interaction might fail because there is a participant missing.

On the other hand, there are often ports on components that do not *need* to be attached, such as a monitoring or logging interface on a database, and there are interactions that can continue even when one of the participants is missing. So we cannot simply outlaw unattached ports or roles.

To detect this kind incompleteness while maintaining as much flexibility as possible, we introduce another test on a configuration. If a port, for example, is unattached, then it must not depend on observing particular events, and it must not expect to be able to initiate any events. In effect, it must be able to behave as the process that simply halts, *i.e.*, §.

Test 10 (Attachment Completeness) *Every unattached port (role) in a configuration must be compatible with the role (port)* §.

As with configuration descriptions, there are many forms of completeness that can be considered for a style description. The most important form of style completeness can obviously not be tested entirely within the WRIGHT formalism, because it depends on information that cannot be captured in any formalism: Does the style description, type definitions and predicates, completely cover the intentions of the style developer? That is, do the constraints exactly include desired systems and exclude undesired systems? This question corresponds roughly to the question of whether a program is "correct." It can only be answered formally in terms of some other requirements specification, and even then it is only as good as that other document.

There is another, simpler, form of completeness that can be addressed within WRIGHT. We can ask whether the style constraints are sufficient to guarantee consistency. That is, can we prove that whenever the style constraints hold (test 9), all other consistency properties also hold (tests 1-8)? If this property, which we will call *consistency-completeness*, holds, then users of the style will find their task greatly simplified.

7 Semantics of WRIGHT

Thus far we have described WRIGHT and its consistency and completeness tests informally. We now describe the formal basis for WRIGHT, elaborating the details of the tests and providing formal definitions for the concepts that were introduced earlier.

7.1 CSP: Traces, Failures, and Divergences

Before we go into the details of WRIGHT's semantics, we need to review the semantics of CSP, the basic notation for behaviors in WRIGHT. CSP is based on the concepts of alphabets, traces, and refusals. The CSP model used here is as described by Hoare [Hoa85].

Formally, a CSP process is modelled as a triple, (A, F, D), where A is the process' alphabet, F is its failures, and D is its divergences.

We have already seen how the alphabet of a process is important to understanding what behaviors a process controls. The alphabet of a process is the set of events in which the process may engage. The alphabet of a process P is often written $\alpha\mathsf{P}$.

The second element of the CSP model of a process is its failures. The failures of a process are pairs of traces and refusals. Each trace is a sequence of events, and each refusal is a set of events. Thus, we write: $\mathit{failures}(\mathsf{P}) \subseteq \mathrm{seq}\,\alpha\mathsf{P} \times \mathbb{P}\,\alpha\mathsf{P}$.

The traces of a process are those sequences of events that are permitted by the process. The process $\mathsf{P} = \mathsf{a} \rightarrow \mathsf{P} \,\Box\, \mathsf{b} \rightarrow \mathsf{P}$, for example, can generate the traces $\langle\rangle, \langle a\rangle, \langle b\rangle, \langle aa\rangle, \langle ab\rangle, \langle ba\rangle$, *etc.* The entire set of traces is indicated by $\mathit{traces}(\mathsf{P})$.

The behavior of a process is not fully determined by its traces. Recall that when describing a process we were careful to distinguish between internal, or non-deterministic choice, and external, or deterministic, choice. In the first case, the environment is expected to offer a set of events to the process, and the process itself controls which of the events will occur. Because the environment can also prevent any event from occurring (by excluding that event from the offered events), we say that a process has the ability to *refuse* any of the alternatives by selecting a different one. If the process uses deterministic choice, on the other hand, it cannot refuse any of the events, because the environment can now force any one of them to happen.

Refusal of events is modelled by the process' failure pairs. The first element of a single failure pair is a trace of the process and the second element is a refusal of the process after it has engaged in that trace.

Notice that a failure is a trace and a *set* of events. Also, there can be more than one refusal for a given trace. This is because a process may be able to refuse some combinations of events separately, but not all in combination. For example, the process $\mathsf{Q} = \mathsf{a} \rightarrow \mathsf{Q} \sqcap \mathsf{b} \rightarrow \mathsf{Q}$ can refuse the event a or the event b, but not both at the same time. Thus, it has failures $(\langle\rangle, \{\mathsf{a}\})$ and $(\langle\rangle, \{\mathsf{b}\})$, but not $(\langle\rangle, \{\mathsf{a}, \mathsf{b}\})$. The process $\mathsf{R} = \mathsf{a} \rightarrow \mathsf{R} \sqcap \mathsf{b} \rightarrow \mathsf{R} \sqcap \mathbf{STOP}$, however, does have the failure $(\langle\rangle, \{\mathsf{a}, \mathsf{b}\})$ because the choice of **STOP** would mean that neither event can occur even if both are offered.[14]

The final part of the CSP process model is its divergences. The divergences are those traces after which the process is equivalent to CHAOS, defined as follows:

$$\mathsf{CHAOS}_A = \mathbf{STOP} \sqcap (\forall\, x : A \sqcap x \rightarrow \mathsf{CHAOS}_A)$$

This process is termed *divergent* because it is the most unconstrained, unpredictable process: It can either refuse or accept any event at any time. The past behavior of the process is no help in predicting its future behavior.

Divergences are used to represent catastrophic situations or completely unpredictable programs (such as those containing infinite loops without any communication events). For simplicity, we will not emphasize the possibility of divergence in our discussion of WRIGHT. None of the definitions and proofs rely on an assumption of absence of divergence, however, so this is not a major omission. For

[14] Formally, we define **STOP** to be the process that immediately refuses all events in its alphabet: $\mathbf{STOP}_A = (A, \{(\langle\rangle, \mathbb{P}\,A)\}, \varnothing)$. The A subscript is omitted when the alphabet is obvious.

example, no divergent process is deadlock-free and therefore we know that no consistent WRIGHT connector or component contains any divergences.

Because we augmented the basic CSP notation with a special annotation for events, to indicate whether they are initiated or observed, we must also augment the CSP model to make this distinction about its alphabet. The subset of P's alphabet which is initiated is α_iP, and the subset that is observed is α_oP.[15]

7.2 CSP Refinement

As we saw previously, when we described tests that apply to WRIGHT specifications, we need a way to compare two processes that are not identical. A CSP process describes a pattern of behavior, and we would like to be able to substitute another behavior in its place if it matches that pattern. We want, in effect, a refinement relationship that guarantees that one process satisfies all of the properties of another, possibly as well as some other properties of its own.

One way of thinking about satisfaction of properties is to ask whether an external observer could possibly tell that one process had been substituted for another. Consider two processes P and Q. Let's try to decide whether P is a refinement of Q. What can we, as external observers, do with P to determine if it is different from Q? We can execute any of the traces that are part of Q's behavior pattern and then offer it different combinations of events. If we are dealing with Q we know that it can only refuse a set of events if it is in the failure set of Q. Q can only accept an event (*i.e* continue a trace) if the new, extended trace is part of its trace set.

Now, suppose P always obeys these constraints, *i.e.* it never refuses a set of events unless that set is in the failures of Q and it never accepts a trace unless it is in the traces of Q. Then we will never be able to detect that the process is P and not Q. Hence, P has all of the properties of Q that we care about: P is a refinement of Q.

Thus, P is a refinement of Q whenever its failures are a subset of Q's failures. Any traces of Q appear in the first element of the failure pair, and so subsetting failures ensures that P's traces are a subset of Q's. The refusals are the second element of the failure pair, and so Q's refusals are also respected by P.

Definition 1 (Refinement) *A process* $\mathsf{P} = (\alpha\mathsf{P}, \mathit{failures}(\mathsf{P}), \mathit{divergences}(\mathsf{P})$ *is a refinement of* $\mathsf{Q} = (\alpha\mathsf{Q}, \mathit{failures}(\mathsf{Q}), \mathit{divergences}(\mathsf{Q}))$, *written* $\mathsf{Q} \sqsubseteq \mathsf{P}$, *if* $\alpha\mathsf{P} = \alpha\mathsf{Q}$ *and* $\mathit{failures}(\mathsf{P}) \subseteq \mathit{failures}(\mathsf{Q})$ *and* $\mathit{divergences}(\mathsf{P}) \subseteq \mathit{divergences}(\mathsf{Q})$.

In the following examples, it is assumed that the alphabet of both P and Q is $\{\mathsf{e}, \mathsf{f}\}$:

1. $\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \sqcap \mathsf{f} \rightarrow \mathsf{Q} \sqsubseteq \mathsf{P} = \mathsf{e} \rightarrow \mathsf{P}$
2. $\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \sqcap \mathsf{f} \rightarrow \mathsf{Q} \sqsubseteq \mathsf{P} = \mathsf{e} \rightarrow \mathsf{P} \,\Box\, \mathsf{f} \rightarrow \mathsf{P}$
3. $\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \sqcap \mathsf{f} \rightarrow \mathsf{Q} \not\sqsubseteq \mathsf{P} = \mathsf{e} \rightarrow \mathsf{P}$

In example 1, process P removes all of the traces involving f from Q. It may do so because the non-deterministic choice in Q means that f is always in the refusal set, even though it is also in the traces. In example 2, process P does not change the traces of Q, but it removes f from the refusals, reducing non-determinism. In example 3, P is *not* a refinement of Q, because it refuses f when Q does not.

[15] Notice that these subscripts mark the locus of control (initiated/observed), *not* the direction of dataflow (input/output). Remember that, typically, input events are observed (and therefore appear in α_o) and output events are initiated (and therefore appear in α_i).

7.3 Using CSP to Model WRIGHT

Having described the formal basis for CSP, we can in turn use CSP as a formal basis for WRIGHT. We show how the notations used in WRIGHT can be combined, first into processes representing component and connector types, and then into a single CSP process representing the entire instantiated system.

Recall that we introduced notations for component types, connector types, instances of components and connectors, and attachments. Component type descriptions are divided into the port specifications and the computation. Connector type descriptions are divided into the role specifications and the glue. Each of these elements is described by a behavior description based on CSP. Also, type declarations can be parameterized.

Now we will shift our use of CSP from a notation to a model. That is, instead of using CSP as a part of the WRIGHT notation, relying on the event declarations and operators, we will use the underpinnings of CSP (alphabets, traces, refusals) to give meaning to the entire WRIGHT notation. That is, for each element of the WRIGHT notation (component type, connector type, attachment declaration, *etc.*), we will define a mapping from that part of the WRIGHT syntax to a corresponding CSP process or operator. We will then show how these elements of CSP are combined to give meaning to an entire WRIGHT description, in effect, providing a denotational semantics for WRIGHT. On the basis of this formal definition, WRIGHT descriptions can be precisely analyzed and we can prove (or disprove) properties of the described system.

Our strategy for giving meaning to a WRIGHT description will be to take the basic CSP processes that represent each of the parts of the description and to combine them via the CSP $\|$ operator. A major hurdle to this is that the semantics of CSP provide for *global* event names. That is, the way that two CSP processes indicate that they interact is by sharing identical event names. If two processes are combined that share event names, then interaction can occur. Conversely, if the processes use different event names, no interaction can occur. Global name matching is not an appropriate model for an architectural notation, because we want to make the communication pathways explicit (by declaring connectors and attachments) so that the parts can be separately structured and analyzed. To accomplish the structuring and explicit declaration of events, we must ensure that event names in WRIGHT declarations act as *local* names. That is, if an event name is declared in two different component types, these do *not* refer to the same CSP event. On the contrary, we must ensure that these are different events, and that the communication occurs only via connectors.

On the other hand, we must ensure that when a communication pathway is specified, with a combination of connector instances and attachments, interaction *does* occur. That is, if a connector instance is attached to a component instance, the corresponding events in their process must result in interaction; at the CSP level, they must represent the same event (by name).

To accomplish this mapping of local WRIGHT events to global CSP events, we use a technique of *event renaming*. This is accomplished via functions that transform one set of event names to another set of event names. These functions can then be applied to a process, resulting in a new process that has the same behavior pattern, but uses different events.

Definition 2 (Renaming) *If f is a function from events to events, and $\mathsf{P} = (A, F, D)$ is a CSP process, then $f(\mathsf{P}) = (A', F', D')$ where $A' = f(\!|A|\!)$, $F' = \{t' : \operatorname{seq} A'; r' : \mathbb{P}\, A' \mid (\exists (t, r) \in F \mid t' = f \circ t) \wedge r' = f(\!|r|\!) \bullet (t', r')\}$ and $D' = \{d' : \operatorname{seq} A' \mid (\exists\, d : D \mid d' = f \circ d)\}$.*

7.4 Component and Connector Types

The basic building block of a WRIGHT description is the type declaration. These specify the behavior patterns that we use to construct the overall meaning of a WRIGHT system description.

Our use of the type descriptions in constructing the behavior of a WRIGHT system instance is quite simple: As we discussed in the informal introduction to WRIGHT, we use only the **Computation** or **Glue** specifications to represent the behavior of the component or connector, respectively. These provide enough information to understand how each part contributes to global behavior. As we discussed in Section 6.1, and will see in more detail in later sections, this does not mean that the role and port specifications are unnecessary. They will come into play when we analyze the type specifications for consistency, and they will help us determine whether a particular attachment (for example) is legal without exhaustive analysis of the system.

Also, the initiate/observe annotations on events are not used in constructing the CSP process that determines a type's behavior. In order to ensure that what one element initiates, another observes, the two versions of an event are considered identical in the CSP behavior process. Recall that these markings are used to check whether a given connector is legal. The distinction will also be kept when checking a **Computation** against its ports (so that a computation may not substitute an initiated event when the port had declared it to be observed, for example).

The only complication in using the **Computation** or **Glue** specification as a CSP process from which we can construct a system's meaning is if there are parameters that are used in the definitions. If the parameter is a *process* parameter, the actual parameter defines a process that has the name of the formal parameter. (*i.e.* if the formal parameter name is P, then the actual parameter is taken to define a process named P.) This definition is then considered part of the definition of the specification process. Any other kind of parameter (*e.g.* a numeric parameter) is interpreted by *textual substitution* of the actual parameter into the component or connector type definition.

7.5 Instances and Attachments

The overall CSP process representing the behavior of a WRIGHT specification is constructed by combining one process for each declared instance in the system. There is one process for each component instance, representing its computation, and one process for each connector instance, to synchronize the communication along the declared paths. Each process is placed in parallel, using the CSP $\|$ operator.

As we indicated in the previous section, the basic pattern for each process is the **Computation** or **Glue** process for the instance's type. This process must then be modified in order to create the desired synchronization patterns. Because CSP uses global event names, we must modify the processes using renaming. As we discussed in Section 5.3, the problem with CSP's global events is two-fold: First, there may be multiple instances of any given type, so there may be undesired interactions if we introduce multiple copies of a given process. Second, the event names used by each component or connector type are local — the events in a **Glue** process use event names based on the role name, while events in a **Computation** process use events based on port names. It is likely that desired synchronizations (*i.e.* those indicated by attachments) will not occur without modifying the base event names.

Instances

To overcome the problem of *undesired sharing* of event names, each process in the CSP model of a WRIGHT system is renamed so that the name of the instance is prefixed to the local event names. For example, a component whose **Computation** refers to an event Port.event in its type definition and is given the instance name C would use the event name C.Port.event instead. This is accomplished through a special kind of renaming, *relabelling*:

Definition 3 (Relabelling) *For a process* P *and a name* L, $\mathsf{L{:}P} = f_L(\mathsf{P})$, *where* $f_L(\mathsf{e}) = \mathsf{L.e}$ *for all events* $\mathsf{e} \neq \surd$, *and* $f_L(\surd) = \surd$.

Relabelling is sufficient to construct processes to represent each component instance. We simply use a relabelled version of the **Computation** associated with the component type. For a declaration "N : CT," where the component type CT has a computation process P, we will use the relabelled process N:P. This has the effect of giving each event of component instance N a three leveled structure: The component name, the port name, and the local event name (N.P.e). If the computation uses any internal events (not associated with any port) these will have two level names: the component name and then the event name (N.e).

Attachments

Each component instance is an independent computation, and it therefore makes sense to isolate one component's events from another, but what about connectors? The purpose of a connector is to provide an interaction pathway. If we simply isolate the events by making them unique to the connector instance process, then no interaction will occur. It will not be enough simply to relabel the connector process. Instead, we must ensure that the event names used in the connector glue are matched to the event names used in the components whose interaction it is describing. Thus, if a connector instance C has a role R, which is attached to a component port N.P, we need to ensure that each event R.e in the component glue is renamed N.P.e, since that is the name used in the component instance.

Before applying any relabeling to cause this desired sharing of events, we must first ensure that there will be no *undesired* sharing of events based on role or event name clashes. As for components, we can use relabelling to avoid undesired sharing of event names for connector instances. All connector type processes are initially relabelled with the name of the connector instance. Thus, our starting point for renaming a connector to ensure interaction paths will be event names of the form Conn.Role.e. The desired effect is that such an event will be renamed Comp.Port.e whenever there is an attachment "Comp.Port **as** Conn.Role".

To achieve this, we use another special form of renaming function:

Definition 4 *For any names* N,N',M, M',

$$\mathcal{R}^{(\mathsf{N},\mathsf{M})}_{(\mathsf{N}',\mathsf{M}')}(\mathsf{e}) = \begin{cases} \mathsf{N}'.\mathsf{M}'.\mathsf{e}' & \textit{if } \mathsf{e} = \mathsf{N}.\mathsf{M}.\mathsf{e}' \\ \mathsf{e} & \textit{otherwise} \end{cases}$$

In the case of the attachment above, we would thus use $\mathcal{R}^{(\mathsf{Conn},\mathsf{Role})}_{(\mathsf{Comp},\mathsf{Port})}$. We will call this function an *attachment function*. In the next section we show how these functions are applied to the connector instance processes to ensure that the behavior model of a WRIGHT configuration uses the communication pathways laid down by the connector instances and attachment declarations.

7.6 Configurations

Now we have all of the parts necessary to model the behavior of a configuration specification. Each component instance declaration has a corresponding process (the relabelled computation of its type). Each connector instance has a corresponding process (again, the relabelled glue of its type). Each attachment declaration has a renaming function (the attachment function). We can now combine these in a single model:

Definition 5 (Configuration Behavior) *If a configuration declares component instances* $\mathsf{Cp}_1{:}\mathsf{CpT}_1$ *...* $\mathsf{Cp}_n{:}\mathsf{CpT}_n$*, where each component type* CpT_i *has computation process* CpP_i*, connector instances* $\mathsf{Cn}_1{:}\mathsf{CnT}_1$ *...* $\mathsf{Cn}_m{:}\mathsf{CnT}_m$*, where each connector type* CnT_i *has glue process* CnP_i*, and attachment declarations with attachment functions* $\mathcal{R}_1$ *..* $\mathcal{R}_k$*, let* $\overline{\mathcal{R}} = \mathcal{R}_1 \circ ... \circ \mathcal{R}_k$*. Then the behavior of the configuration is the CSP process* $(\forall\, i : 1..n \| \mathsf{Cp}_i : \mathsf{CpP}_i) \parallel (\forall\, j : 1..m \| \overline{\mathcal{R}}(\mathsf{Cn}_j : \mathsf{CnP}_j))$.

7.7 Style Constraints

In Section 3.4 we introduced the concept of style-specific constraints in WRIGHT. These are used to control the properties of any system in that style, and can be used to guarantee that a system has particular properties or that it can be constructed using specialized techniques. The constraints that we discussed before were *syntactic*: They referred to the topology of instances and attachments or constrained elements based on the named types.

Using the underlying CSP model, we can extend our notion of style to permit *semantic* constraints on systems. For example, some styles may depend on a particular relationship existing between two different ports on each component. A database component might have a logging port that records all incoming communication, for example. We could certainly describe a database component type that logs all transactions, by providing a particular database protocol and enumerating all of the messages that would go into the log, but this constrains the components too much: The style may not care about the specific database protocol, but only about the fact that logging will occur. The style specifier should be able simply to *require* that data be logged, without indicating how it is achieved. A possible specification could look like this:

$$\forall\, \mathsf{C} : \mathit{components}; \mathsf{e} : \alpha_o \mathsf{C}; t : \mathit{traces}(\mathsf{C}) \mid \mathsf{e} = \mathit{last}(t) \bullet \{\mathsf{f} : \alpha\mathsf{C} \mid t\langle f \rangle \in \mathit{traces}(C)\} = \{\overline{\mathsf{Log.log!e}}\}$$

This specification states that whenever a component C observes an event e, it will immediately record that event on port Log. It does *not* constrain the component in any other way: The component may have any number of other ports, and the alphabets on those other ports are unconstrained.

In order to support these semantic constraints, we add the following predicates to the constraint language described in Section 3.4. In each case, we use an element (component, connector, port, or role) and its behavior process interchangeably. Constraints refer to processes *before renaming.*

- $\alpha\mathsf{P}$: the alphabet of process P.
- $\alpha_i\mathsf{P}$: the subset of $\alpha\mathsf{P}$ that is initiated.
- $\alpha_o\mathsf{P}$: the subset of $\alpha\mathsf{P}$ that is observed.
- *Traces(*P*)*: the traces of process P.
- *Failures(*P*)*: the failures of process P.
- *Refusals(*P*)*: $\{(\langle\rangle, r) : \mathit{Failures}(\mathsf{P}) \bullet r\}$ (the *immediate* refusals of P).

The use of semantic constraints will be illustrated in the case studies, in later chapters.

7.8 Formal Specifications of Tests

Our final task to complete the formalization of WRIGHT is to elaborate on the formal definition of the tests from Section 6. While we will restate some of the tests formally, for others we will define those parts of the test which were left informal (such as the definition of deadlock-freedom). In each case, the definitions in this section together with the statement of the test in Section 6 will formally define the test.

Port/Computation Consistency

The first test is for Port/Computation Consistency. As we described in Section 6.1, the purpose of this test is to ensure that the ports, viewed as partial specifications of the component, are consistent with the **Computation**, which we view as a more complete specification. The basic idea of the test was to ensure that the port was a projection of the computation into its alphabet.

Recall from Section 3.1 that the port specification had two purposes: As a *requirement* on the component and as an *assumption* about the environment. The requirement is that the component fulfill the behavior described by the port. This is tested by projecting the component into the port's alphabet and testing whether it is a refinement. The assumption is about what the environment (*i.e.* the other parties of the connector to which it is attached) will do during the interaction.

How can we test the requirement aspect of a port while taking advantage of the assumptions? If we ignore the assumptions in the requirements test, many reasonable computations will be excluded; the test will be too strict. We must separate out the assumptions indicated by the ports and allow the computation to take advantage of them.

The key to this separation is that the assumptions that a component makes about event behavior at its ports are based on the *observed* events in the port process. Any initiated events are declarations about what the component will do, not assumptions about the environment. Thus, if our test limits its comparison of the behavior of the component and the port to those traces that match the observed event patterns of its ports, we will have covered all of the requirements of the port while taking advantage of its assumptions.

Formally, we can use a *deterministic* version of a process to restrict a second process to the traces of the first. If we have two processes (with identical alphabets), P and Q, which operate in parallel — $\mathsf{P} \| \mathsf{Q}$ — then the combined process will have no traces that are not also a trace of Q. If Q is *deterministic*, then any non-determinism in the combined process will correspond to non-determinism in P. That is, 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 that is prevented by Q, and no internal choices will be introduced as a result of the interaction with Q.

Thus, for any process Q, if we can construct a process $det(\mathsf{Q})$ but that has exactly the same traces as Q that is deterministic, then a process $\mathsf{P} \| det(\mathsf{Q})$ will have at most the traces of Q, but all of the decisions will be made by P.

So, to model the component's **Computation** process computing in the environment indicated in the ports assumptions, we must do two things: First, we must take the ports and construct a process that is restricted to the pattern of *observed* events. This extracts out the assumptions portion of the port specification. Second, we must take this new process and make it *deterministic*. This ensures that we are testing the decision-making of the **Computation** specification, not the non-determinism of the ports. These constructions are defined as follows:

Definition 6 *For any process* $\mathsf{P} = (A, F, D)$ *and event set* E, $\mathsf{P}\lceil E = (A \cap E, F', D')$ *where* $F' = \{(t', r') \mid \exists (t, r) \in F \mid t' = t \lceil E \wedge r' = r \cap E\}$ *and* $D' = \{t' \mid \exists\, t \in D \mid t' = t \lceil E\}$.

In this definition, a trace projection ($t \upharpoonright E$) indicates a trace which contains all of the elements of t that are in the set E, in the same order, without any of the elements that are not in E. Thus $\langle acadbcbc \rangle \upharpoonright \{a, b\} = \langle aabb \rangle$.

Definition 7 *For any process* $\mathsf{P} = (A, F, D)$, $det(\mathsf{P}) = (A, F', \varnothing)$ *where* $F' = \{(t, r) \mid t \in traces(\mathsf{P}) \wedge \forall e : r \bullet t\langle e \rangle \notin traces(\mathsf{P})\}$.

The function $det(\mathsf{P})$ has the same traces as P, but it has fewer refusals. In fact, it has only those refusals that are necessary to make it a consistent process (*i.e.* it refuses events that do not correspond to permitted traces). Thus, which event occurs at any point is fully controllable by the environment: $det(\mathsf{P})$ is deterministic.

We use the projected, deterministic port processes to interact with the **Computation**. We can then test this restricted form against each port, to see if the **Computation** meets the requirements of the port. The refinement test ensures that the internal decisions (as modelled by non-determinism) match those indicated by the port process specification.

Test 1 (Port/Component Consistency) For a Component with computation process **C** and ports P and $\mathsf{P}_1 \ldots \mathsf{P}_n$, **C** is consistent with P if:

$$\mathsf{P} \sqsubseteq (\mathbf{C} \| \forall i : 1 \ldots n \| \ det(\mathsf{P}_i \upharpoonright \alpha_o \mathsf{P}_i)) \upharpoonright \alpha \mathsf{P}$$

Deadlock Freedom

Tests 2 and 3 are for consistency in a connector. These two tests seek to identify situations where a group of parties to the interaction disagree on how the interaction should proceed or where one of the participants might drop out of the communication in the middle. The two tests are based on freedom from deadlock. A process is said to deadlock if it refuses all events without having terminated (*i.e.* having engaged in $\surd$). Deadlock usually arises because interacting processes disagree about what event should occur next. This is the kind of deadlock that we want to avoid when we test for deadlock in the **Glue**/Role composition. Deadlock in a directly-specified process (*i.e.* one that does not involve the composition of distinct processes) usually indicates one of two things: either an error by the specifier (the specification does not describe the intended computation) or an internal error in a computation (for example, a catastrophic situation that should not arise). By testing the roles in isolation, we avoid this internal deadlock.

We say that a process is deadlock-free if it can never deadlock. That is, the process must always either be willing to continue its computation or be in a condition of *successful* termination.[16]

Definition 8 (Deadlock-Freedom) *A process* $\mathsf{P} = (A, F, D)$ *is deadlock-free if for every trace* t *such that* $(t, A) \in F$, $last(t) = \surd$.

Initiator Rules

There are two rules for connectors involving the distinction between initiated and observed events. The first, called "single initiator," takes advantage of the initiated/observed distinction to ensure that it is clear at every stage of the protocol which party is responsible for continuing the interaction. The rule states that there must be a unique party, either a role or the glue, that initiates each event:
Test 4 (Single Initiator): A component with glue process **G** and role processes $\mathsf{R}_1 \ldots \mathsf{R}_n$, has single initiators if $\alpha_i \mathbf{G}, \alpha_i \mathsf{R}_1, \ldots, \alpha_i \mathsf{R}_n$ partition the set $(\alpha \mathbf{G} \cup \alpha \mathsf{R}_1 \cup \ldots \cup \mathsf{R}_n) \setminus \{\surd\}$. Further, it must be the case that $\alpha_i \mathbf{G} \cap \alpha_o \mathbf{G} = \varnothing$ and $\forall j : 1..n \bullet \alpha_i \mathsf{R}_j \cap \alpha_o \mathsf{R}_j = \varnothing$.

[16] Recall that $\S = \surd \rightarrow \mathbf{STOP}$, and so in effect we require that the only stopped process is $\S$.

The second rule, "initiator commits," ensures that the initiated/observed distinction is meaningful. The intent is that an initiator of an event acts as the single cause of the event, with the final decision of whether or not that event will occur. (Of course, its decision is informed by the protocol up to that point. We simply mean that the initiating process does not share immediate control over that one event.)

Formally, the initiator commits rule is enforced by requiring that the process refuse all other events whenever it chooses to engage in an initiated event:

Test 5 (Initiator Commits): A process $\mathsf{P} = (A, F, D)$ obeys initiator commits if for every trace t and event $\overline{\mathsf{e}} \in \alpha_i \mathsf{P}$ such that $t\langle\overline{\mathsf{e}}\rangle \in \mathit{traces}(\mathsf{P})$, $(t, A \setminus \{\overline{\mathsf{e}}\}) \in F$.

The problem of initiator-commitment, and this formal statement of the rule, is complicated by the possibility of *internal concurrency*. Suppose that a component (for example) is implemented as two independent threads. One of the threads carries out a computation and then initiates an event at one port. Meanwhile, the other thread is waiting to respond to an event that it will observe on another port. In this case, the component does not really control which of the two events will occur first; there is a race condition between the computation of the one component thread and the observation of the event in the other. It should therefore be considered reasonable for a component to violate the strictest form of initiator-commits described above, *provided that each sequential sub-process of the computation* does *obey it.*

Concurrent components with a clear substructure that can be analyzed for initiator-commits will arise, for example, when the architecture of a system is simplified by encapsulating some part of it as a larger component. In this case, the computation of the new component will be described in WRIGHT as a system configuration.

Compatibility

In order to understand the compatibility test, we must understand the relation between a port and a role. The port is a specification of the component, as it is seen from the point of view of a single interaction. That is, it describes a part of the *actual* behavior of the component. The role, on the other hand, acts as a placeholder representing the range of potential participants in the interaction described by the connector. That is, the role also represents a component, but a *potential* component, rather than an actual component.

In its job as description of a potential computation, the role does two things. First, it is a specification of the range of decisions and behavior traces that the component may have. As such, this is similar to the job of the port, but for a class of components rather than a single component. Second, it circumscribes the behaviors over which the connector's rules are expected to apply. That is, it limits the scope of the interaction to those situations that could actually arise given the constraints of the other participants in the interaction.

Consider an analogy to an abstract data type's preconditions and postconditions. A stack type has (say) two methods, push and pop. At first glance, any combinations of pushes and pops are possible, and the methods' postconditions describe what will happen regardless of whether they alternate, all pushes come before all pops, or there are no pushes at all, but only pops. The preconditions narrow the specification of the data type to a more constrained situation, however: *there must be no more pops than pushes.* If this rule is broken, then the data type is not guaranteed to work. That is, the type definer may *assume* that this condition is met, and the type user must *assure* that it is met.

The traces of a role act in a similar manner to the preconditions of the abstract data type. When the role process describes traces that vary among a set of observed events, then the component that

fills the role may assume that one of those observed events will occur, and no others. Even if the component specifier chooses to describe its behavior outside of these constraints (just as the stack definer may indicate what happens if an empty stack is popped), the actual behavior will always fall within the constraints. It is the job of the other participants in the connection to ensure that it does (and the other rules, notably connector deadlock-free and initiator-commits, enforce this). Thus, the effective specification of a component in an interaction is the port process *restricted to the traces of the role process.*

Recalling that trace restriction is handled through the deterministic version of a process, we therefore test the process $\mathsf{P} \| det(\mathsf{R})$ for compliance to the role specification. There is only one additional complication before we can apply the refinement test: The refinement test only applies over processes with identical alphabets. To solve this, we *augment* each process with the missing events:

Definition 9 *For any process* P *and event set* A, $\mathsf{P}_{+A} = \mathsf{P} \| \mathbf{STOP}_A$.

We can now define compatibility with complete precision:
Test 8 A port P is compatible with a role R, written "P **compat** R," if

$$\mathsf{R}_{+\alpha \mathsf{P} \setminus \alpha \mathsf{R}} \sqsubseteq \mathsf{P}_{+\alpha \mathsf{R} \setminus \alpha \mathsf{P}} \| det(\mathsf{R}).$$

8 Conclusion

In this report we have presented the WRIGHTArchitectural Specification Language. To summarize, the main features of WRIGHTare its support for precise description of architectural structures and their abstract behavior, the ability to define architectural styles, and a set of consistency and completeness checks for architectural description.

Throughout the description we have focused primarily on the nature of the language itself—its key features, analytic potential, and semantics. Descriptions of the application of WRIGHTto specific systems and styles can be found in related reports and papers.

Current work on WRIGHTis centered around three activities. First, we are carrying out several industrial case studies. The most substantial of these is a specification of the High Level Architecture (HLA) for distributed simulation. Second, we exploring new formal issues with WRIGHT, including architectural refinement and reconfigurable architectures. Third, we are developing WRIGHTtoolset. Currently under development are tools to translate WRIGHTinto ACME **??**, a converter from a subset of WRIGHTRapide [LAK+95], and a graphical browser for WRIGHTspecifications.

Acknowledgements

Many of our colleagues, both within CMU and in the software architecture research community, have helped clarify our ideas about the nature of architectural specification. We would especially like to thank Tony Hoare, Daniel Jackson, Jeff Kramer, Robert Monroe, Ralph Melton, Mark Moriconi, Mary Shaw, Zhenyu Wang, and Jeannette Wing.

This paper extends work published previously 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.

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

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

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

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

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

[SG96a] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SG96b] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

A WRIGHT BNF

```
BNF for Wright

SpecList := Spec | SpecList Spec;

Spec := Configuration | Style;

Style := "Style" Name
                TypeList
         "Constraints"
                ConstraintList
         "End Style";

TypeList := Type | TypeList Type | null ;

Type := Component | Connector | InterfaceType;

Component := "Component" SimpleName [ '(' FormalParams ')' ]
                         PortList
                         "Computation = " ProcessDescription;

Connector := "Connector" SimpleName [ '(' FormalParams ')' ]
                         RoleList
                         "Glue =" ProcessDescription;

PortList := Port | PortList Port | null ;

Port := "Port" ProcessName '=' ProcessExpression;

RoleList := Role | RoleList Role | null ;

Role := "Role" ProcessName '=' ProcessExpression;

ConstraintList := ConstraintExpression
| ConstraintList ConstraintExpression
| null;

Configuration := "Configuration" Name
                         "Style" Name
                                 TypeList
                         "Instances"
                                 InstanceList
                         "Attachments"
                                 AttachmentList
                         "End Configuration";
```

```
InstanceList := Instance | InstanceList Instance | null ;

Instance := NameList ':' TypeUse;

AttachmentList := Attachment | AttachmentList Attachment | null;

Attachment : Interface "As" Interface;

Interface : Name '.' Name;

TypeUse := SimpleName [ '(' ActualParams ')' ];

InterfaceType := InterfaceypeHead Name  '='  Expression;

InterfaceTypeHead := "Interface Type" | "Process" | "IntType";


Name := Identifier [ '{' ActualParams '}' ]
ProcessName := ProcessID [ '{' ActualParams '}' ]

ProcessDescription :=  ProcessExpression
     | CompositeProcess;

CompositeProcess := Configuration
                    "Bindings"
                          BindingList
                    "End Bindings"

BindingList := Binding | BindingList Binding | null;

Binding := Interface "As" Name;

FormalParams := FormalParam | FormalParams ';' FormalParam;

FormalParam := NameList ':' ParamTypeExpression;

ParamTypeExpression := "Interface Type"
                         | "Process"
                         | "IntType"
                         | "Computation"
                         | "Glue"
                         | "Port"
                         | "Role"
                         | "Component"
                         | "Connector"
                         | "Attachments"
                         | "Integer"
```

```
                        | RangeExpression
 | SetExpression;

ProcessExpression := "Tick"
                        | ProcessExpression ';' ProcessExpression
                        | ProcessExpression "->" ProcessExpression
                        | ProcessExpression "Where" DeclList
                        | ProcessExpression '||' ProcessExpression
                        | ProcessExpression '[]' ProcessExpression
                        | ProcessExpression '|~|' ProcessExpression
                        | "forall" FormalParams '[]' ProcessExpression
                        | "forall" FormalParams '|~|' ProcessExpression
                        | "forall" FormalParams ';' ProcessExpression
                        | "forall" FormalParams '||' ProcessExpression
 | EventName '?' VarName
                        | EventName '!' VarName
                        | ProcessName
                        | "Computation" [ '(' Name ')' ]
                        | "Glue" [ '('  Name ')' ]
 | ProcessName ':' ProcessExpression
                        | '(' ProcessExpression ')';

VarName := Name;

EventName := EventName '.' EventName | Name;

LogicalExpression :=
                          "not" LogicalExpression
                        | LogicalExpression "or" LogicalExpression
                        | LogicalExpression "and" LogicalExpression
                        | "forall" FormalParams '.' LogicalExpression
                        | "forall" FormalParams '|' LogicalExpression '.'
LogicalExpression
                        | "exists" FormalParams '.' LogicalExpression
                        | "exists" FormalParams '|' LogicalExpression '.'
LogicalExpression
                        | Term "==" Term
                        | Term "!=" Term
                        | Term '<' Term
                        | Term '>' Term
                        | Term '<=' Term
                        | Term '>=' Term
                        | '(' LogicalExpression ')'
                        | Term "in" ParamTypeExpression
                        | Term "not in" ParamTypeExpression;

ConstraintExpression := LogicalExpression;
```

```
Term := Name | '(' Term ',' Term ')' | ProcessName | '(' Expression ')';


SetExpression :=
                          SetExpression 'union' SetExpression
                        | SetExpression 'intersection" SetExpression
                        | SetExpression 'setminus' SetExpression
                        | SetExpression 'cross' SetExpression
                        | 'power' SetExpression
                        | '{' ActualParams '}'
                        | '{' FormalParams '|'  LogicalExpression [ '.'
LogicalExpression]  '}'
                        | '{' FormalParams [ '.' LogicalExpression] '}'
 | '(' SetExpression ')'
                        | SetExpression '^' SetExpression;

MathExpression :=
                          MathExpression '+' MathExpression
                        | MathExpression '-' MathExpression
                        | '(' MathExpression ')'
                        | Term;

RangeExpression :=
                          MathExpression ".." MathExpression
                        | MathExpression ".."
                        | ".." MathExpression;

Expression :=    ProcessExpression
               | SetExpression
               | MathExpression
               | LogicalExpression;

AnyName := ProcessName | Name;

DeclList := Declaration | DeclList Declaration;

Declaration := AnyName '=' Expression [ "When" LogicalExpression ];

NameList:= Name;

SimpleName := Identifier;

ProcessID : String;

ActualParams := Expression | ActualParams ',' Expression | null;
```

B A Complete Example

We define a pipe-filter system that capitalizes every other character read from an input stream. The system consists of three filters: Split, Capitalize, and Merge. The first sends alternative characters to the second filter which capitalizes them. The third merges the two streams.

The definition is in two parts. First we define the PipeFilter Style. Then we define the specific system in that style.

B.1 The PipeFilter Style

Style PipeFilter

Interface Type DataOutput = $(\overline{\text{write!x}} \rightarrow \text{DataOutput}) \sqcap (\overline{\text{close}} \rightarrow \S)$
Interface Type DataInput = $(\overline{\text{read}} \rightarrow (\text{data?x} \rightarrow \text{DataInput} \,\square\, \text{end-of-data} \rightarrow \overline{\text{close}} \rightarrow \S)) \sqcap (\overline{\text{close}} \rightarrow \S)$

Connector Pipe
Role Source = DataOutput
Role Sink = DataInput
Glue = $\text{Open}_{\langle\rangle}$
where $\text{Open}_{\langle\rangle}$ = Source.write?x$\rightarrow\text{Open}_{\langle x\rangle}$
$\square$ Source.close$\rightarrow$Closed$_{\langle\rangle}$
$\square$ Sink.close$\rightarrow$Capped
$\text{Open}_{s\,\hat{}\,\langle x\rangle}$ = Sink.read$\rightarrow\overline{\text{Sink.data!x}}\rightarrow\text{Open}_s$
$\square$ Source.write!y$\rightarrow\text{Open}_{s\,\hat{}\,\langle x\rangle\,\hat{}\,\langle y\rangle}$
$\square$ Source.close$\rightarrow\text{Closed}_{s\,\hat{}\,\langle x\rangle}$
$\square$ Sink.close$\rightarrow$Capped
$\text{Closed}_{s\,\hat{}\,\langle x\rangle}$ = Sink.read$\rightarrow\overline{\text{Sink.data!x}}\rightarrow\text{Closed}_s$
$\square$ Sink.close$\rightarrow\S$
$\text{Closed}_{\langle\rangle}$ = Sink.read$\rightarrow\overline{\text{Sink.end-of-data}}\rightarrow$Sink.close$\rightarrow\S$
$\square$ Sink.close$\rightarrow\S$
Capped = Source.write!x$\rightarrow$Capped
$\square$ Source.close$\rightarrow\S$

Constraints
$\forall\, c : \mathit{Connectors} \bullet \mathit{Type}(c) = \text{Pipe}$
$\forall\, c : \mathit{Components}; p : \mathit{Port} \mid p \in \mathit{Ports}(c) \bullet \mathit{Type}(p) = \text{DataInput} \vee \mathit{type}(p) = \text{DataOutput}$

End Style

B.2 The System

Configuration EveryOther
Style PipeFilter

Component SplitFilter(nout: 1..)
Port Input = DataInput
Port $\text{Output}_{1..nout}$ = DataOutput

Computation = $\mathsf{Transfer}_1$
where $\mathsf{Transfer}_i$ = $\overline{\mathsf{Input.read}} \rightarrow (\mathsf{Input.data?x} \rightarrow \overline{\mathsf{Output}_i\mathsf{.write!x}} \rightarrow \mathrm{Transfer}_{i+1}$
$\Box\ \mathsf{Input.end\text{-}of\text{-}data} \rightarrow \mathsf{Close})$
$1 \leq i \leq nout$
$\mathsf{Transfer}_{nout+1} = \mathsf{Transfer}_1$
$\mathsf{Close} = \overline{\mathsf{Input.close}} \rightarrow \forall\, i : 1..nout\ ;\ \overline{\mathsf{Output}_i\mathsf{.close}} \rightarrow \S$

Component MergeFilter(nin: 1..)
Port $\mathsf{Input}_{1..nin}$ = DataInput
Port Output = DataOutput
Computation = $\mathsf{Transfer}_1$
where $\mathsf{Transfer}_i$ = $\overline{\mathsf{Input}_i\mathsf{.read}} \rightarrow (\mathsf{Input}_i\mathsf{.data?x} \rightarrow \overline{\mathsf{Output.write!x}} \rightarrow \mathrm{Transfer}_{i+1}$
$\Box\ \mathsf{Input}_i\mathsf{.end\text{-}of\text{-}data} \rightarrow \mathsf{Close})$
$1 \leq i \leq nin$
$\mathsf{Transfer}_{nin+1} = \mathsf{Transfer}_1$
$\mathsf{Close} = \overline{\mathsf{Output.close}} \rightarrow \forall\, i : 1..nin\ ;\ \overline{\mathsf{Input}_i\mathsf{.close}} \rightarrow \S$

Component UpperFilter
Port Input = DataInput
Port Output = DataOutput
Computation = $\overline{\mathsf{Input.read}} \rightarrow$
$((\mathsf{Input.data?x} \rightarrow \overline{\mathsf{Output.write!}(cap(\mathsf{x}))}) \rightarrow$ **Computation**
$\Box\ (\mathsf{Input.end\text{-}of\text{-}data} \rightarrow \overline{\mathsf{Input.close}} \rightarrow \overline{\mathsf{Output.close}} \rightarrow \S))$

Instances
Split:SplitFilter(2)
Merge:MergeFilter(2)
Upper:UpperFilter
P_1, P_2, P_3 : Pipe

Attachments
$\mathsf{Split.Output}_1$ **as** P_1.Source
Upper.Input **as** P_1.Sink

$\mathsf{Split.Output}_2$ **as** P_2.Source
Merge.Input2 **as** P_2.Sink

Upper.Output **as** P_3.Source
$\mathsf{Merge.Input}_1$ **as** P_3.Sink

End Configuration

C Mechanical 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 automate the tests that WRIGHT defines for well-formed architectures.

For suitably constrained subsets of WRIGHT this is possible. In particular, if one restricts process notation so that only a finite number of processes are used, then it is generally possible to use tools for exhaustive state exploration (such as model checkers) [BCM+90] to verify properties of the processes and to check relationships between processes. In particular, we can use the emerging technology of automated verification tools for process algebras, such as using FDR [FDR92].

FDR is able to check whether one process is a refinement of another. Thus provided we can express our tests as checks of the form $P \sqsubseteq Q$, for appropriately constructed finite processes P and Q, we can use the tool.

To illustrate, suppose we wish to check compatibility between a port

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

and a role

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

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: These are encoded in FDR[17] as:

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

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

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

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 $STOP_{\{\mathrm{set}\}}$:

```
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
```

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

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

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

The translation process that we have just illustrated is relatively straightforward to automate. We have done so with using the 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.[18]

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 non-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 by 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 "$e \rightarrow (P(e) \Box Q(e)) [] f \rightarrow (P(f) \Box Q(f)) ...$"

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, other properties such as port-computation consistency, and connector deadlock-freedom can be checked by tools such as FDR.

[18] 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.