A Formal Approach to Software Architecture

Robert J. Allen

May 1997

CMU-CS-97-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee

David Garlan (chair)
Daniel Jackson
Mary Shaw
Barry Boehm (USC)

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

This research was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330; by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031; by the Defense Modeling and Simulation Office; and by National Science Foundation Grant No. CCR-9357792. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the US Department of Defense, Wright Laboratory, Rome Laboratory, the United States Government, or the National Science Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Keywords: Software Architecture, Formal Specification, Architecture Description Languages, Software Architectural Style, WRIGHT

Abstract

As software systems become more complex, the overall system structure—or software architecture—becomes a central design problem. A system's architecture provides a model of the system that suppresses implementation detail, allowing the architect to concentrate on the analyses and decisions that are most crucial to structuring the system to satisfy its requirements.

Unfortunately, current representations of software architecture are informal and ad hoc. While architectural concepts are often embodied in infrastructure to support specific architectural styles and in the initial conceptualization of a system configuration, the lack of an explicit, independently-characterized architecture or architectural style significantly limits the benefits of software architectural design in current practice.

In this dissertation, I show that an Architecture Description Language based on a formal, abstract model of system behavior can provide a practical means of describing and analyzing software architectures and architectural styles.

This dissertation demonstrates this claim through WRIGHT, an architectural description language based on the formal description of the abstract behavior of architectural components and connectors. WRIGHT provides a practical formal basis for the description of both architectural configurations and of architectural styles. It is distinguished by the use of explicit, independent connector types as interaction patterns, the ability to describe the abstract behavior of components using a CSP-like notation, the characterization of styles using predicates over system instances, and a collection of static checks to determine the consistency and completeness of an architectural specification. We introduce techniques to support the analysis of large-scale systems, and demonstrate WRIGHT's expressiveness and practicality through three case studies.

To Katrina and William

Acknowledgements

It hardly seems possible to me that I have finally reached the end of this long process, and that it only remains to thank the people who have helped me along the way. Even beyond the formal aspects of my education, I have learned much and gained much from the people around me over the last seven years, and I know that I will never be able to truly express my appreciation. I can only say: Thank you!

To David, I say: Thank you! You have taught me about research, about teaching, speaking, and writing, about software architecture, and about life. You have supported me at every stage, from my first arrival here at CMU to the final completion of my dissertation. You have supported me when I was coasting along not getting much accomplished and when I was pushing hard to do more than seemed possible; when I was here in Pittsburgh and when I was away in Durham. You have truly gone beyond the call of duty.

To Mary, Daniel, and Barry, I say: Thank you! You have helped make my dissertation the best that I can make it. You have been patient with me when I had nothing to give you and responsive when I gave you too much all at once. Your feedback and your example have enriched my experience as well as my work.

To the ABLE group, I say: Thank you! You put up with my half-baked ideas and my buggy prototypes and turned them into something real and even useful.

To the SSSG, I say: Thank you! You have made CMU a rich, exciting environment in which to do research in software engineering. Your insightful and constructive feedback has helped to improve all of the work I have done here. I know that if you like it, every audience will be easy.

To the CMU SCS community, I say: Thank you! Your caring, openness, and support has made my tenure here a pleasant one. You have always been there to help make things possible and have bent over backwards to smooth my path. You have always been reasonable, and often more than reasonable.

To my office mates, past and present, I say: Thank you! You have tolerated my muttering and teasing and aggressive work-avoidance tactics. You have helped me to know that it can be done, and to remember that there is more to computer science than my little corner of it.

To Lynette, Heather, and Matthew, I say: Thank you! You have always, and especially during my visits from out of town, been welcoming and accommodating.

To my family, and especially Katrina, I say: Thank you! Your support has kept me going when I thought I couldn't go on. You have kept me in perspective, and helped me keep my priorities straight. You are what makes it all worthwhile.

Thank you all! — *Rob*

Contents

Chapter 1

Introduction

1.1 Motivation

An important problem facing software developers is the increasing size and complexity of software systems. As the expectations of users of software increase, software developers are expected to produce software to handle more difficult problems on a larger scale.

As the complexity of software systems increases, the overall system structure—or software architecture—becomes a central design problem. Software architecture provides a model of the large scale structural properties of systems. These properties include the decomposition and interaction among parts as well as global system issues such as coordination, synchronization, and performance.

The software architecture of a system often appears in system descriptions as a "boxes and lines" diagram. This diagram structures the system in terms of particular kinds of computations and their composition. For example, the architecture of a payroll system might decompose it into three parts: a database, a report generator, and a data entry front end. These parts appear as boxes in an architectural diagram. Lines connecting them indicate the use of queries and updates supported by the database. (See figure 1.1.)

Software architecture raises the level of abstraction at which developers can reason about their systems. A system's architecture provides a model of the system that suppresses implementation detail and increases the independence of system components, permitting many issues to be localized. By suppressing these details at the architectural level, the architect can concentrate on the analyses and decisions that are most crucial to the system structure.

Architectural Interaction

A critical issue in software architecture is composition. Once a system has been decomposed into components, they must be re-composed to define the structure of the system as a whole. An important class of composition in software architecture is active interaction between components based on discrete actions. Components each carry out some part of the total computation and interact to combine their behaviors, resulting in a behavior for the system as a whole. Interactions can be quite simple, such as in a batch model where each component acts separately, one executing to completion, its output providing the input to

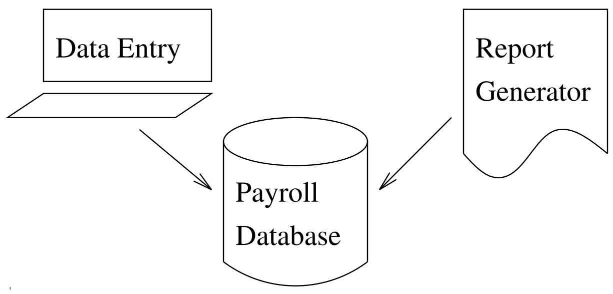

Figure 1.1: Boxes and lines diagram: A payroll system.

another component, which executes in a separate phase. Interactions can also be quite complex, such as network protocols of distributed systems, where each component can initiate communication, generate messages, and respond to other components' messages, where buffering, reliability, and authentication of information passed between components must be taken into account.

In order for components participating in these interactions to achieve overall system goals, not only must the interfaces of components be consistent, but components must agree on the ordering of actions, what party is responsible for what part of the interaction, whether a component's behavior is optional or required, and so on. Further, it must be clear whether any mechanisms external to the components are required to support the interaction.

At the same time, particular patterns of interaction tend to recur in a given system, and across different, but similar, systems. Whether it be batch processing with shared files, parallel systems with shared data, signal processors with streams of continuous data, or reactive components with event broadcasts, different components are composed into a functioning system using the same kinds of interactions again and again. The goal of an architectural model of a system is to exploit these and other patterns in the system structure—if we can take advantage of these patterns, recognizing where commonalities exist in a system's structure and behavior while making the distinctions necessary to understand the critical properties of the architecture, the effective complexity of the system can be reduced while at the same time our ability to analyze and manipulate the system's design is increased.

Architectural Style

Another important aspect of software architecture is the extension of design to exploit commonalities across families of systems. When developing a particular system, designers tend not to explore all possible alternatives for its architecture. Instead, they use specific patterns and idioms that are effective for the domain in which they are working. These patterns and idioms constrain the design space, permitting developers to ignore complications and alternatives that are not relevant to the system that they are developing. This exposes the issues that are most important and thus helps the developer make effective choices and locate the best solution more easily. We term such a collection of patterns and idioms an architectural *style*.

To motivate the idea of style, consider the familiar example of a compiler. The first compiler required years of development effort by a substantial team. Many alternatives had to be considered for the system because no one had built anything like it. Today, however, the principles of compiler construction are well understood. In particular, a compiler's structure is expected to include a lexical analyzer, a parser, a collection of semantic checks (such as a type checking), and a code-generating back end. The problem has been focused on these specific pieces, construction techniques have been standardized, and tools have automated many routine aspects of compiler implementation. As a result, a compiler can now be developed by one undergraduate in a single semester. Elements such as code optimizers can be developed, often separately from the rest of the compiler, and added easily. New techniques and technology are made available with a minimum of additional development effort.

Although the compiler is an unusually well-developed example, use of architectural style is ubiquitous in software development. Most high level design descriptions include some characterization of the architectural style of the system, using phrases such as "it is a pipe and filter system," "we use a client-server architecture," "the system is based on a blackboard."

Using a style has many benefits. A style focuses the design problem on techniques that are effective for a specific class of systems. By recognizing that, for example, real-time considerations are not of interest to a payroll database, developers can instead concentrate on developing a flexible and general set of queries for the database. A collection of components and connectors that work within a style enhances flexibility and reuse. Chimera [SVK93], for example, enables the developer of a robot controller to reuse many components by providing standard mechanisms for combining them while maintaining effective real-time performance. The use of particular models supports higher-level design abstractions. If a style guarantees that a set of properties hold, it can lead to more powerful analyses than a general architecture permits.

1.2 Problems with State of the Art

Unfortunately, with few exceptions current exploitation of software architecture and architectural style is informal and ad hoc. While architectural concepts are exploited in infrastructure to support architectural styles and in the initial conceptualization of a system configuration, the lack of an explicit, independently characterized architecture or architectural style significantly limits the extent to which software architecture can be exploited using current practices.

Currently, architectural configurations are typically described using informal box and line diagrams in design documentation, providing little information about the actual computations represented by boxes, their interfaces, or the nature of the interactions between them (represented by lines).

This lack of information severely limits the usefulness of these diagrams. Without specific information about the interfaces that are expected between components, and without enough information to determine the meaning of a composition of different elements, implementors are forced either to guess the intentions of the architect, essentially redoing the design work, or to continually consult with the original designer, making that person a bottleneck in the development effort and creating significant source of risk.

One way that developers have attempted to provide more precise descriptions of architectures is through the use of implementation constructs such as objects or shared communications infrastructure. An architecture might be realized, for example, as a "layered" implementation, where lower layers implement infrastructure to support particular kinds of interactions, while higher layers correspond to architectural components. In this model, the module structure, defined in a programming language such as Standard ML [HMM86] or Ada [DoD83] can be taken to represent the "architecture" of the system. However, when interpreted this way, calls that appear to communicate with modules at lower layers of the infrastructure may actually represent indirect, abstract communications with other components at the same level as the caller.

The absence of an abstract definition of the interactions within a layer means that overall system behavior presented in the architecture is difficult, if not impossible, to analyze. Abstract behaviors must first be extracted from the many concrete details of the lower layers of infrastructure, and only then can system behavior be meaningfully considered. The abstract behavior of components and their interactions is still not explicitly defined, and so analysis can not be carried out at an abstract level.

For example, suppose we have a system that performs a series of data transformations, and that the details of the system's interactive behavior are described in terms of calls to an input/output implementation library. If the abstract architectural description provides a throughput figure for each of the components, what can we determine about overall system throughput? Unfortunately, we can say nothing more than that it is *probably* no better than the worst single component in the system. There are so many different interpretations and implementations of data input/output systems that a composition of the individual figures to construct an aggregate throughput can not be justified without extensive analysis of the actual code underlying the architecture.

When component interactions are, at most, informally specified, it is also more difficult to reuse high-level design effort in other systems. There is no way to determine whether an architecture or pattern of composition may be appropriate in a different context, because it is not clear which properties of a system are intrinsically part of its architecture and which properties are incidental to the architecture, arising from how the architecture is interpreted in the implementation. When the architecture is bound into an implementation structure it becomes more difficult to extract and consider the architecture independent of the full details of the system in which it was constructed. The architectural assumptions of a component or communication infrastructure element are not explicit and so it is risky and often impossible to move that element to another system.

Current approaches to architectural style are also informal and ad hoc. The architectural style of a system, if one is identified at all, often appears as a simple phrase characterizing the system. A system description might include a catch-phrase such as "pipe-and-filter" or "centralized database." These descriptions appeal to the developer's (or other reader's) intuitive understanding of a style and do not have any precise definition behind them.

A style might be known by a prototypical example or a recognizable topology. For example, many data transformation systems have a "pipeline" topology. The architecture of these systems might be considered examples of the pipeline style.

Sometimes a specialized notation supports a style. These notations include ways of composing a system from parts or of specifying a component or connector. For example, CORBA's Interface Definition Language [OMG91] has been developed to specify interfaces to distributed objects. CORBA IDL can be viewed as a language for specifying connectors in distributed object systems. Other examples of style-specific models of system construction include the POLYLITH Software Bus [Pur94] and FIELD's event-based tool integration [Rei90].

Styles also appear as a library of routines or other run-time system. For example, the Task Control Architecture [SLF90] provides a collection of procedures to support communication in a robot controller.

For individual styles, some progress has been made on combining notations and tool support specifically at the architectural level. Examples include architectures and tools developed under the Domain Specific Software Architecture program [DAR90].

All of these approaches to style are informal. Although they may have well-defined tools (such as a notation or run-time system), the architectural implications of the style lack a precise characterization.

This lack of a precise characterization means that communicating the meaning of the style to others is difficult. Often, only the originators of a style or a few "gurus" understand the style in depth. This limitation reduces the utility of the style. In the absence of a well defined style the architecture of a system drifts away from that style and becomes difficult to understand or support.

The lack of a precise formal basis for style also limits the analytic leverage of a style. It is essentially impossible to prove properties of an informal style or of a system architecture that refers to such a style. In addition, there is no way to confirm that a system's implementation corresponds to an informal architectural description.

1.3 Partial Solutions

As we have seen, the current informal practice of the description of configurations and styles limits the extent to which we can take advantage of software architecture in the design and analysis of software systems. Analysis of configurations and component interaction is hampered either by the absence of a precise specification of a system's architecture or by having it encoded in terms of implementation details. Exploitation of style is hampered by the inability to precisely characterize a style and to reason about the relation of individual systems to that style.

Progress has been made on providing a more sound, practical basis for describing and reasoning about software architecture. Current suggested solutions to this problem fall into two categories, which we will consider in this section. The first kind of solution is to apply existing formal methods to architectural design, and the second is to develop notations, architecture description languages, to supplement or replace informal boxes-and-lines diagrams. Each of these approaches has important benefits. At the same time, each has weaknesses that mean that the approach cannot, alone, provide a sufficient basis for architectural description and analysis.

1.3.1 Formal Methods

One possible solution to the ad hoc state of the practice in architecture is to apply general formal methods to modelling and analyzing architectures and architectural styles. By providing a precise semantics for the system at the abstract level of architecture, a formal model of the system can provide the basis for rigorous, justifiable analysis of critical system properties. If formal modelling is applied at the level of style, analyses can be extended to any system that conforms to the style. For example, in [AG92], we used Z to formally prove that any system in the pipe-filter style can be encapsulated as a filter in another, larger pipe-filter system.

A formal model of the architecture can also be used as a basis for verification of an implementation. Because the constraints that must be met by a system are precisely defined, it is possible to determine whether a system conforms to an architecture and whether a given architecture conforms to a style. Also, a formal model can expose exactly the abstractions that are of importance, without binding other implementation details, and thus the implementation can retain the flexibility to alter non-architectural decisions without violating the architecture or requiring that all analyses (from the architectural level) be redone.

By raising the level of abstraction at which the semantics of a system are defined, a formal description of an architecture has the potential to increase the architecture's effectiveness as a vehicle for communication about a system's design. The nature of the compositions and the role of each component in a system's functionality are highlighted by reducing the amount of detail in the description. Thus, a properly structured and abstract formal description can retain much of the simplicity of a "box-and-line" description while providing much more information about the actual character of the system.

Unfortunately, the application of a general formal method to the description of software architecture and architectural styles is not a trivial undertaking. The very generality of formal methods means that, at a given level of abstraction, all of the architectural structures and semantics must be laid out before a system or family of systems can be analyzed.

The lack of pre-defined structure of a general formalism means that the architectural abstractions must be defined from first principles for each new description. For example, architectural descriptions are based fundamentally on the principle of localization, but a formalism such as CSP provides no means of localizing the effect of an interaction so that it is independent of the global context in which it is placed. Also for example, a formalism such as Z provides little in the way of a computational model; thus, to provide a model of the communication patterns of a system or style, one must be constructed from basic sets and relations.

At best, a formal model of a system or style "reuses" the intuitions of the model's developers from their previous experience. There is no common framework upon which the structural patterns and abstractions of an architectural description can be based.

Because there is no common framework, each style developer must invent his or her own models and tools to support architectural abstractions. Developing these models is costly. As a result, the models are often elaborated only as much as the original developer needs. This means that the abstractions used to structure the system may be ill-understood, inadequately supported, and, worse, lost or forgotten as the implementation of a system evolves. The use of these ad hoc formalizations does not provide the abstraction and understanding of the system for which it was developed.

1.3.2 Purely Structure-based Architecture Description Languages

Another approach to supporting architectural description and analysis is through Architecture Description Languages, or ADLs. These languages provide notations for decomposing a system into components and connectors and specifying how these elements are combined to form a configuration.

By providing a direct realization of architectural abstractions as constructs in a descriptive notation, ADLs permit architects to expose and define the structure of their systems. This provides a means of communicating more effectively for the purposes of evaluating the design, for considering how it might be adapted to another use, and for guiding detailed design and implementation of the system.

ADLs, through the use of large scale constructs and the encapsulation of different parts of a system, are well-suited to the description of real systems. They permit the architect to select the level of granularity at which the system will be described and to encapsulate different parts of the system in separate, hierarchically structured descriptions.

As a central, abstract description of the system that can be tied directly to implementations, an explicit architectural description can serve as a central focus for tracing different aspects of the system. These aspects include allocation of requirements, division of development responsibility and effort, changes to the system over time, and incremental or dynamic construction.

If tools are provided to support the notation, a description of a system or style in an ADL can be used to generate more reliable implementations at lower cost, by reducing the amount of new code that must be developed and increasing reuse (through the selection of appropriate implementations by the tools). Also, style-specific analysis can be exploited when a system configuration is described using style-specific vocabulary and tools. For example, the MetaH notation provides a complete environment to support the development and implementation of avionics control systems [WKE93].

Typically, however, ADLs provide no direct means of specifying properties of individual elements such as the behavior of a component or the pattern of interaction represented by a connector.[1] This means that there is little or no analytic leverage to be gained from these notations.

A particular ADL may support system construction (*i.e.*, compilation and runtime instantiation) by making implementation units correspond to components and the use of particular infrastructure for interaction. However, this does little to inform the architect about what properties his or her system will have, whether it will meet its requirements, or even if the description is internally consistent.

Some ADLs do provide significant analytic leverage (*e.g.*, MetaH). But they do so by fully defining certain aspects of the system (typically interaction semantics). These constraints mean that the range of systems that can be described in the notation is significantly limited. In effect, these ADLs actually define one particular style, allowing configurations

[1] There are some exceptions, such as Rapide. These are discussed in chapter 2.

in that style to be described, but not permitting the description of configurations outside that style or the description of any style. Thus, while they provide significant leverage when that particular style is appropriate and the design is known, from the beginning, to be in that style, they can not provide any support outside the domain for which they were developed. It also is not possible to use the notation to describe and analyze families of systems, rather than a single system at a time.

1.4 A Balanced Approach: Formal Basis for Architectural Description

These two approaches to software architecture have complementary strengths and weaknesses. Formal methods provide strong analytic power, abstraction, and independence from implementation details, but they have the weakness that they fail to provide basic architectural constructs and abstractions, such as connectors and styles, directly. This leads to the separate development of these concepts for each new description and an inability to get results in a cost-effective manner.

ADLs, on the other hand, provide a much better match to architecture by defining notations for components and connectors that are ready to apply to the problem of constructing complex real-world systems. Currently, however, ADLs are weakened by exactly those things that make formal methods attractive: The ADLs are either weak in analytic power, overly-specific to a subset of systems of interest, tied to specific implementations, or express the architecture at an overly concrete level of detail.

What we need instead is a way of combining these two approaches: To proceed by recognizing the structure and abstractions of architecture, to make the description tractable while providing a formal basis for reasoning about the properties of the system. This avoids the problem of intractability and impracticality of a general formal method while providing the benefits that are unavailable in a semantics-free or implementation oriented architecture description language.

In this dissertation I show that:

> *An Architecture Description Language based on a formal, abstract model of system behavior can provide a practical means of describing and analyzing patterns of discrete, asynchronous interaction in software architectures and architectural styles.*

Our demonstration of this claim will be through the example of a particular formal architecture description language, WRIGHT, which models connector types as abstract patterns of interaction using a CSP-like notation, and defines architectural styles by predicates constraining configurations. WRIGHT connector types define an abstract model of the relation between discrete, asynchronous component actions, modelling patterns of interactions independent of particular component instances. In addition to showing the utility and practicality of WRIGHT, we will further suggest how this model can be extended to other kinds of interactions and architectural connections such as synchronous communication and global data-dependent properties.

1.5 Requirements for Description and Analysis of Architecture

In order to provide a practical basis for description and analysis of architecture, a notation or model must support the following:

- *Description of architecture configurations:* The ultimate goal of architectural description is to capture and exploit a system's structure at a high level of abstraction. It must be possible to define the components that will occur in a system and the interactions between them.

- *Description of architectural styles:* Beyond describing single systems, architects should be able to delineate the members of a *family of systems*. These architectural styles provide a means of exploiting commonalities between systems and for leveraging analyses and implementation efforts. Additionally, the two kinds of description should be able to be combined — given a configuration, we want to know if it is a member of a given style.

- *Analysis of properties of interest:* An important goal of the descriptive activity is to use the description to understand properties of the system that has been or will be constructed. The architect must be able to use a description to analyze the system or style to determine whether it satisfies its requirements. Note that it is important to analyze both individual configurations and architectural styles. Ultimately, it is the properties of individual systems that are important, but analysis of styles provides important leverage by applying to more than one architectural configuration.

- *Application to practical problems on real systems:* While *some* analytic leverage is arguably an improvement over no ability to analyze a description, a notation is not generally practical if it can only be applied in tightly constrained circumstances or only on small systems. A practical notation must address this by scaling to apply to complex, real-world problems.

By exhibiting a language that meets these requirements, we can demonstrate the utility of our approach, combining the power of formal methods with the utility of Architectural Description Languages.

1.6 The WRIGHT Architecture Description Language

In the remainder of this dissertation we introduce WRIGHT, an architectural description language based on the formal description of the abstract behavior of architectural components and connectors, to show how it can meet the above requirements. WRIGHT provides a practical formal basis for the description of both architectural configurations and of architectural styles. It is distinguished by the use of explicit, independent connector types as interaction patterns, the ability to describe the abstract behavior of components using a CSP-like notation, the characterization of styles using predicates over system instances, and

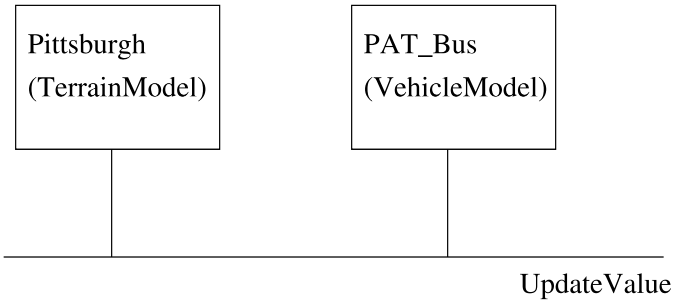

Figure 1.2: Diagram of example architecture.

a collection of static checks to determine the consistency and completeness of an architectural specification. Because the semantics of WRIGHT specifications are formally defined, an architecture characterized in WRIGHT provides a sound basis for reasoning about the properties of the system or style described.

1.6.1 Architectural Configurations

To give a sense of WRIGHT and its capabilities, we now now illustrate the main ideas using a simple example. The example is a simplification of the case study described in chapter 6.

Consider a simple system to simulate a bus driving through Pittsburgh. One possible architecture is shown in figure 1.2. One component models the bus and its movements, while another maintains information about the environment through which the bus travels. The two components communicate by transmitting updates of the values of objects' attributes. Figure 1.3 shows the outline of how this would be expressed in WRIGHT. This figure shows the main elements of a configuration description: the use of explicitly described components and connectors, and the delineation of instances and their attachments. The basic configuration corresponds to the diagram of figure 1.2, but provides constructs for the definition of computational elements and their combination via connectors. In our example, the terrain model Pittsburgh is accessed by the vehicle PAT_Bus, using the UpdateValues interaction pattern.

In WRIGHT, the description of a component has two important parts, the *interface* and the *computation*. A component interface consists of a number of *ports*. Each port defines a separate interaction in which the component will participate. Both components in our example have only a single interaction. The Pittsburgh component will provide map value updates via the ProvideMap port, while the PAT_Bus component will interact with its environment via the Environment port.

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters, while an RPC connector represents one component requesting a service of another. A WRIGHT description of a connector consists of a set of *roles* and the *glue*. Each role defines the behavior of one participant in the interaction. A pipe has two roles, the source of data and the recipient. In our example system, the C connector has two roles, one for each Model that it will coordinate. The connector glue defines how the roles will interact with each other.

Each part of a WRIGHT description – port, role, computation, and glue – is defined using

Configuration SimpleSimulation
  Component TerrainModel(*map* : **Function**)
    Port ProvideMap = *[Interaction Protocol]*
    Computation = *[provide terrain data]*
  Component = VehicleModel
    Port Environment = *[Interaction Protocol]*
    Computation = *[compute vehicle movement]*
  Connector UpdateValues(*nsims* : 1..)
    Role $\text{Model}_{1..nsims}$ = *[Interaction Protocol]*
    Glue = *[Data travels from one* Model *to another]*
 Instances
  Pittsburgh : TerrainModel(*[map of Pittsburgh]*)
  PAT_Bus : VehicleModel
  C : UpdateValues(2)
 Attachments
  Pittsburgh.ProvideMap, PAT_Bus.Environment **as** C.Model
End SimpleSimulation.

Figure 1.3: A simple simulation.

a variant of CSP [Hoa85]. For example, the Model role of UpdateValues might be defined by:

$$\textbf{Role}\ \text{Model} = \overline{\text{update!x}} \rightarrow \text{Model} \\ \sqcap\ \overline{\text{request}} \rightarrow \text{newValue?y} \rightarrow \text{Model} \\ \sqcap\ \S$$

This defines a participant in an interaction that repeatedly either provides an updated value ($\overline{\text{update!x}}$) or requests a new value ($\overline{\text{request}}$). If it requests a new value, it will be provided one (newValue?y). It may also choose to terminate successfully at any time (§).

Simply by making the meaning of an architectural description precise, WRIGHT can help the architect communicate his or her ideas to others. For example, the Model role defines exactly what actions a component may or may not take if it is to participate in an UpdateValues interaction. While this precision is valuable simply in terms of description, it also provides a basis for analyzing architectures. As we will discuss in later chapters, descriptions of connectors can also be used to determine whether the connector satisfies certain critical properties. These properties include the internal consistency of the protocol and whether the roles are sufficiently constrained to ensure proper behavior by participants. In considering the UpdateValues connector above, for example, we notice that, as it is described, if both PAT_Bus and Pittsburgh were to choose to request a value before providing an update, a conflict will occur. Both expect a value and there is no value available. In addition to analyzing connectors, components can be analyzed to determine, for example, whether they conform to their interface specifications.

WRIGHT further structures the description of an architectural configuration by distinguishing between component or connector types and specific instances of them in the configuration. In our example, UpdateValues is a connector type: it is defined by a set of potential participants, the Models, and constrains how they may behave, via the Glue. C is an instance of this type: it has two particular participants, Pittsburgh and PAT_Bus, which are associated with the protocol in the attachments (PAT_Bus through port Environment, will play one of the Model roles in the interaction, for example).

The global behavior of a WRIGHT architecture *system instance* is constructed from the processes introduced by the component and connector types. This is done by suitable renaming of events so that a component's events are communicated via the connectors to which it is attached. In particular, renaming causes the glue of a connector to mediate the interactions between the components – effectively enforcing its protocol on the communication.

By deriving the global system behavior from the architectural structure and behavior descriptions of types, WRIGHT provides a means of extending the type-level guarantees to system instances. At the configuration level, WRIGHT provides checks to confirm that a given component port properly fulfills the obligations of any role to which it is attached. If the appropriate constraints are met, then any analyses at the type level automatically apply to instances.

1.6.2 Architectural Style

In addition to describing and analyzing system configurations, WRIGHT permits the designer to describe and analyze entire families of systems, or architectural styles. As we discussed above, by formalizing a style the architect is able to leverage analysis across many systems and thus reduce the effort to produce individual systems.

A WRIGHT style has two parts: (1) vocabulary and (2) constraints on configurations. Vocabulary is introduced by defining component and connector types, just as in a configuration. Constraints define predicates which must be true of any configuration in that style.

For example, the simple simulation of a bus in Pittsburgh shown above can be viewed as one member of a general class of simulations that are coordinated via an UpdateValues connector. We can define a style to delineate this family as shown in figure 1.4. This style provides the UpdateValues connector, so that it can be used in all configurations in this style, as well as a generic interface definition that potential models may use. In addition, the style indicates that there will be exactly one form of interaction, and that it will be via an UpdateValues connector. Because there are no constraints placed on components, a configuration in the style is free to use any collection of components that can participate in the UpdateValues connector. Both the number and the computation of the components can vary.

In later chapters we will show how WRIGHT supports the practical analysis of styles both for consistency and for special properties of interest to architects that might use the system. For example, we will show in a fuller elaboration of the simulation style that the omission of a parameter from the start-up routines of individual simulations leads to situations where there can be no consistent global notion of a "paused" simulation.

Using the Simulation style, we can restate the original simple simulation configuration. This new description, shown in figure 1.5, is simpler because it doesn't need to redefine

```
Style Simulation
  Interface Type SimInterface = [Interaction of one simulation]
  Connector UpdateValues(nsims : 1..)
    Role Model_{1..nsims} = SimInterface
    Glue = [Data travels from one Model to another]
  Constraints
    ∃ C : Connectors | {C} = Connectors
                     ∧ Type(C) = UpdateValues
End Style.
```

Figure 1.4: A simulation style.

```
Configuration SimpleSimulation2
  Style Simulation
    Component TerrainModel(map : Function)
      Port ProvideMap = SimInterface
      Computation = [provide terrain data]
    Component = VehicleModel
      Port Environment = SimInterface
      Computation = [compute vehicle movement]
  Instances
    Pittsburgh : TerrainModel([map of Pittsburgh])
    PAT_Bus : VehicleModel
    C : UpdateValues(2)
  Attachments
    Pittsburgh.ProvideMap, PAT_Bus.Environment as C.Model
End SimpleSimulation2.
```

Figure 1.5: A simple simulation (revised).

the vocabulary introduced by the style.[2] Also, because its relation to the larger family of systems is now explicit, architects can take advantage of results for the style when analyzing the individual system.

1.7 Summary of Contributions

Beyond the overall contribution of demonstrating the practicality of a formal approach to architectural description and analysis, the contributions of this thesis fall into four main categories: the general foundations of software architecture, description and analysis of software architectures, the value of WRIGHT specifically, and contributions to formal methods generally. In each category, the contributions are as follows:

[2]While this example is not much simpler, because of its small size, we can see that the relative amount of simplification grows with the size and complexity of the system being described.

- To Architectural Foundations
  - Explicit connectors – an improved understanding of connector types as patterns of interaction;
  - Notion of style as a predicate over configurations;
  - Example of providing alternate architectural semantics based on architectural structure.
- To Architectural Analysis/Description
  - Abstract behavior as a semantic basis for architectural description – beyond structure, but using structure;
  - Technique of encapsulation/structural induction as a reasoning principle in families of systems;
  - Coverage of both instances and styles in the same formalism.
- Specific Benefits of WRIGHT
  - An Architecture Description Language that provides precise, practical, architecture descriptions and the ability to analyze them;
  - Consistency/completeness checks beyond syntax and name matching, with automated tools to check them;
  - Results of "HLA" case study are useful to developers of that style.
- To Formal Methods in General
  - An example of domain specialization;
  - Consideration of families of systems.

1.8 Plan of Dissertation

The next chapter of this dissertation surveys research related to this dissertation. Then, in chapter 3 we introduce WRIGHT in detail. This is the key chapter for those who want an introduction to WRIGHT and the standard checks, as it explains the language and its constructs as well as discussing the intuitions behind its use.

The next three chapters demonstrate the value of WRIGHT through three case studies. The first, chapter 4, describes the AEGIS system and shows an extended description and analysis of a configuration. The second, through the "Justo-Cunha" style, demonstrates the analytic leverage to be gained by considering styles. The third, chapter 6, explores the "High Level Architecture for Simulations" and shows how WRIGHT can be applied to large-scale systems. It will also provide some specific techniques for handling the need to balance the amount of effort against the benefit of achieving a given level of analytic leverage.

In chapter 7 we discuss the automation of WRIGHT's consistency and completeness checks using commercial model checking tools. Chapter 8 shows an example of extending WRIGHT descriptions for analysis using semantic models not based on CSP.

In chapter 9 we provide a critical evaluation of WRIGHT, asking whether it does indeed meet the requirements described above. We will thus show the practical applicability of a formal architectural description language to the task of analyzing and describing architectural interconnections and styles.

Finally, we conclude in chapter 10 with a summary of contributions and a discussion of future work.

Chapter 2

Related Work

2.1 Introduction

In this chapter we present an overview of related work. We begin with a brief overview of the origins of the field of software architecture, and then discuss work in two categories that are most closely related to our work: The application of general formal methods to architectural analysis and the development of Architecture Description Languages. We include such tools as Aesop, which provide the means of describing architectures but are not strictly languages, in Architecture Description Languages. We will further discuss work in programming languages as well as other kinds of system description languages such as Module Interconnection Languages that relates to the work of this dissertation. Additionally, we will discuss object-oriented design notations, as well as some efforts to support architectural analysis that are not tied to a specific representation.

2.2 Conceptual Background

A conceptual basis for software architectural was originally laid out by two groups, Shaw and Garlan, and Perry and Wolf. In this early work, the main issues and conceptual building blocks, such as components, connectors, configurations, and styles, are described. These motivational papers [PW92, GS93, SG94, SG95] introduce the concept through informal examples, and do not attempt to supply specific mechanisms or notations. Rather, they identify an emerging field and call for research in the area.

Shaw and Garlan [GS93, SG94, SG95] describe software architecture as a necessary step in raising the level of abstraction at which software is conceived and developed. They explain that, just as the informal development of knowledge about data types in the sixties led to its codification as abstract data types in the seventies, so developers are starting to achieve an informal notion of module integration and architectural design, and this will lead to its codification as software architecture. They describe a collection of architectural styles, such as pipe-filter systems and blackboards, to show that this kind of abstraction is found informally in practice and to provide an example of design trade-offs in the selection of styles.

Shaw and Garlan present a model of architecture based on two abstractions: the *com-*

ponent, an independent unit of computation, and the *connector*, an interaction among components. Shaw and Garlan suggest that architectural styles form around component and connector types, repeating patterns of computation and interaction, together with rules for how these are used in specific configurations.

Perry and Wolf [PW92] describe an overall view of software architecture as a mediator between requirements and design. They espouse an abstract view of architecture as a triple: $(Elements, Form, Rationale)$. They justify this through a view of the architectural process as the successive application of constraints to a design. They also introduce the idea of style as constraints on a class of architecture; however, they do not make a clear distinction between instances and styles. For them an architectural configuration also consists of a collection of constraints, and so the division between a configuration and a style is blurred.

While our model of style as constraints on configurations is compatible with Perry and Wolf's view of style, our model of configurations is more closely aligned with that of Garlan and Shaw. We will make a clear distinction between styles, which are defined via *constraints* on architectures, and configurations, which instantiate architectural types in a specific topology. Variability within a single configuration, rather than begin modelled via sets (as is a style's variability), appears as non-deterministic behavior or abstraction of details.

2.3 Formalization of Architecture

Efforts to formalize software architecture began with efforts to apply formal methods directly to individual systems or styles. More recently, there have been some more general treatments of architectural description and analysis. Our discussion of formalization begins with the characterization of individual systems and styles.

2.3.1 CHAM

Most uses of general formal notations in architectural description have focussed on the description and characterization of architectural styles.[1] One exception is the work of Inverardi and Wolf [IW95].

In this work, the CHAM (Chemical Abstract Machine) model [BB92] is applied to the description of the architecture of a compiler. This formalization describes the structure and abstract behavior of a single configuration, rather than a class of systems. Inverardi and Wolf argue that CHAM provides a useful basis for the description of architecture because of its ability to compose specifications from parts and to explicitly describe computation rules.

However, because this work has, to date, been presented in the context of a small set of example systems, it does not provide a general means of formalizing architectures. Any structuring that takes place in a system model must be constructed from scratch for that system, with little or no means of extending analysis beyond the individual system and the properties of that system that are originally laid out. Further, because the formalization

[1] This is probably because efforts on characterizing individual configurations have primarily taken an ADL approach.

is unique and hand-crafted, there are no clear criteria for analyzing the consistency or completeness of the configuration description.

2.3.2 Formalizations of Individual Styles

There have been a number of efforts to apply formal methods to the characterization of architectural styles. These efforts have used Z [Spi92] as their notation, and describe the structure and semantics of a single class of system. Examples include formalization of a class of signal processing applications [DG90], event systems [GN91], the pipe-filter style [AG92], and an object-composition standard [SSM97].

These formalizations have the advantage of applying to a class of systems rather than a single system, and thus have the potential to apply their analysis to multiple systems. They establish properties that systems in the described style share as well as precisely delineating membership in the style.

However, these stand-alone efforts have significant shortcomings that limit the practicality of this approach: First, each is described in isolation, and the abstractions and techniques that they share by virtue of describing architectures must be re-invented for each formalization effort. Thus the cost of developing such a formalization is increased and the availability of tools is limited. Second, because they develop the semantics of a style from scratch, these formalizations provide no direct means of characterizing a specific system configuration. This means that there are only informal means of comparing any configuration description with the style and thus taking advantage of the style's properties. Third, again because they are developed independent of a general architectural model, it is not clear how to establish important properties such as the style's consistency or completeness.

Our approach differs from these efforts in providing a single, common architectural framework in which to model the semantics of a style. By providing an architectural framework and vocabulary, we make it easier to describe and analyze a new style. This is because the common architectural abstractions are already available, and only the unique, style-specific definitions must be introduced. Additionally, by using an architectural notation, we provide a more natural means of expressing the architectural concepts that form a style.

2.3.3 Generalization of Style in Z

Observing that many individual style formalization efforts shared a common set of characteristics, Abowd *et al.* [AAG93, AAG95] define a common framework in Z for the definition of an architectural style. This framework specifies that a style is defined by three parts: A concrete, *style-specific vocabulary* for the description of instances, a *semantic model* for reasoning about systems, and a set of *interpretation mappings* from the configuration syntax into the semantic model. These mappings provide interpretations for component, connector, and configuration descriptions.

By providing a common framework for formalizing style, Abowd *et al.* are able to compare styles that share a common semantic model and to provide guidance to developers regarding the steps that are necessary to formalize a style. Thus, while the effort to formalize a style has not necessarily been reduced, it is at least clear what is involved.

Their common model has a number of advantages over stand-alone style characterizations: It clarifies the relationship between a configuration and style, by showing how a style defines a semantic model (*i.e.* a collection of related Z schemas) and a configuration is an element of that model (*i.e.* an instance of one of the schemas). It provides an interpretation of configuration consistency (its description must be in the domain of an interpretation mapping). It permits some styles to be related to each other (if they share a semantic model, then their intersection can be computed). In effect, this work contributes to the *theory* of software architecture by providing a possible model for what a style is, and how a style model relates to descriptions of configurations.

However, this approach has three main disadvantages that limit its practical application: (1) The cost of developing a style, (2) the use of unique semantic models, and (3) the dependence on style-specific syntax. The first disadvantage arises directly from disadvantages (2) and (3). Because there is no common framework for semantics, there is no less effort involved in describing a style using the techniques of Abowd *et al.* than there is for the stand-alone efforts described above.

The use of unique, style-specific syntax and semantics leads to two difficulties: First, there is no way to describe and analyze a system that is not a member of a style. Because all decisions about descriptive form and semantics are tied to a style, it is necessary to decide what style is to be used for a system *before it can be described*. Because the semantics are style specific, if a system is not a member of a style, it cannot be assigned any meaning in the given framework. It is not feasible to develop a style by generalizing from a collection of system examples, because the examples can not be precisely characterized without the style.

Our approach is consistent with the theory of Abowd *et al.* but with a crucial difference: We allow a style to introduce new vocabulary for use in configurations, *within the framework of a common notation*. That is, we permit a style developer to *augment* an existing syntax and semantics. Thus, while Abowd *et al.* provide a framework for understanding the relation of an architectural style to description and analysis of configurations, we provide more direct support for style specific description and analysis. Our notation does this by starting with a common syntactic and semantic basis, and providing notations for characterizing a style and its configurations. The style developer is not required to construct special interpretation mappings or underlying semantic models, because our model provides them already.

2.3.4 Moriconi *et al.*

The style model of Moriconi *et al.* [MQ94, MQR95] is similar to that of Abowd *et al.* with two main differences: They use first-order logic for the definition of both configuration and style structures, and they provide a specific model of style-based refinement of configurations.

Like Abowd *et al.*, they define a style through style-specific vocabulary and semantics, and also like Abowd *et al.*, the definition of a configuration is in terms of that style-specific vocabulary. Instead of providing interpretation mappings that map distinct configuration entities into a style's semantic space, however, they define a configuration as a collection of predicates in first order logic, which refer to the atomic predicates defined and related by a style. Thus, Moriconi *et al.* view a configuration as a logical theory that *includes* a style's theory. Like Abowd *et al.*, however, a configuration can not be interpreted independent of the axioms provided by the style, as the style gives meaning to the terms used in the configuration's axioms.

The most important difference between the work of Abowd *et al.* and Moriconi *et al.* is the latter's focus on a regular model of refinement. Moriconi *et al.* define refinement through interpretation mappings from one logical theory (*i.e.* a configuration or a style) to another. They use semantic mappings to relate styles and then reuse the semantic mappings, together with name mappings between instances, to relate configurations.

In this model, a refinement mapping is considered legal if the image of the concrete theory under the mapping is exactly equivalent to the abstract theory. That is, all properties that are provable in the abstract theory must be provable in the concrete theory, and all properties that are expressible but *not* provable in the abstract theory must not be provable in the concrete theory, as mapped into the vocabulary of the abstract theory. This strict rule defines a refinement relation, rather than true equivalence, because the concrete theory is allowed to add additional properties as long as they occur only in vocabulary that is not mapped into the vocabulary of the abstract theory (and thus can not be used in any proofs in the abstract theory).

Because they use a similar style definition model, where a unique syntax and semantics are developed for every style, Moriconi *et al*'s model suffers from the same weaknesses as that of Abowd *et al.* Each style must re-define its own constructs for structuring configurations and provide compositional rules for the meaning of configurations of elements.

Additionally, because a style must define a complete, self-contained theory, and because configurations are also interpreted as theories, reasoning about configurations and their conformance to a style is complicated. Moriconi *et al* require a completeness assumption about their specifications. The completeness assumption states that all properties not provable in a theory are false. This assumption means that all proofs of refinement or style membership are required to make an argument about the weakness of their specification as well as the much simpler demonstration of property satisfaction. This further means that if an architect wishes to permit a property to be un-specified in a given system description, he or she must select a semantic theory in which it is *impossible* to state that property. Otherwise, failure to specify a property supplies a default answer. It is simpler, and more typical, in formalization to select a fixed specification language and to use *non-determinism* to indicate when a given property is not constrained. This is the approach we have taken. In this approach, if a property cannot be proved either to be true or false in a specification, it is assumed that a refinement may make either choice. Moriconi's completeness assumption complicates reasoning because the language used must be modified whenever the specifier wishes to leave a given decision unbound. The distinction between these approaches is similar to the distinction between *initial*, *terminal*, and *loose* semantic interpretations in algebraic specification, as discussed in, for example [Wir90].

2.3.5 Reflexion Model

Another technique that can be used to study the relation between different specifications has been developed by Murphy *et al.* [MNS95]. Given a pair of graphs, one representing an "abstract" specification and another a "concrete" specification, and a mapping between them (of nodes to nodes), they compute a "reflexion model." The reflexion model identifies the relation between edges in the two graphs. For any pair of nodes, there are four situations: An edge is present in the abstract graph but not the concrete, an edge is present in the concrete graph but not the abstract, an edge is in both graphs, or there is no edge in either graph. Examples of abstract graphs include architectural configurations, object-class diagrams, entity-relation diagrams, *etc.* Examples of concrete graphs include module dependency graphs and data- and control-flow graphs automatically derived from source code.

Murphy's reflexion model does not provide a model of architecture, but can be used to take advantage of one if it is defined. For example, a reflexion model could be used to consider the relation of an abstract architectural description (in our notation, for example) and a system's implementation.

2.4 Architecture Description Languages

Recently there have been proposed a number of Architecture Description Languages. These languages provide notations for describing the structure of software systems in terms of hierarchical configurations of interacting components. These languages provide an explicit and common basis for describing architectural configurations. Thus they do not suffer from the problems discussed in the previous section that arise from the requirement that structures be re-invented for each new system. However, as we will see, current ADLs have three major weaknesses: Lack of support for style, lack of a basis for architectural analysis, and insufficient ability to describe architectural interactions (connectors).

2.4.1 Darwin

Darwin [MDEK95, MK96] is an architectural description language developed by Magee and Kramer. It describes a component type by an interface consisting of a collection of services that are either provided (*i.e.* declared by that component) or required (*i.e.* expected to occur in the environment). Configurations are developed by component instantiation declarations and bindings between required and provided services. Darwin supports the description of dynamically reconfiguring architectures through two constructs — lazy instantiation and explicit dynamic constructions. Using lazy instantiation, a logically infinite configuration is described and components are instantiated only as the services they provide are used by other components. Explicitly dynamic structure is provided through the use of *imperative* configuration constructs. In effect, the configuration declaration becomes a program that is executed at run-time, rather than a static declaration of structure.

Darwin provides a semantics for its structural aspects through the π-calculus [MPW92]. Each provided service is modelled as a channel name, while each binding declaration is a process that transmits the name of that channel to a component that requires the service. Magee and Kramer have used this model of structural elaboration to analyze the distributed algorithm that they use in the implementation of Darwin configurations.

In a Darwin-generated implementation, each primitive (non-hierarchical) component is assumed to be implemented in some programming language, and platform specific glue

```
component pipeline (int n) {
  provide input;
  require output;

  array F[n]:filter;
  forall k:0..n-1 {
    inst F[k];
    bind F[k].output -- output;
    when k<n-1 bind
      F[k].next -- F[k+1].prev;
  }
  bind
    input -- F[0].prev;
    F[n-1].next -- output;
}
```

Figure 2.1: A pipeline in Darwin[MK96, p. 5].

code is generated for each service type. The elaboration algorithm acts, essentially, as a name-server that provides the location of provided services to any executing components.

Despite the presence of a π-calculus model for Darwin's structural descriptions, Darwin does not provide an adequate basis for analysis of the behavior of an architecture. This is because the model does not provide any means of describing the properties of either a component or its services. Component implementations are uninterpreted black boxes, while the collection of service types is a platform-dependent collection whose semantics is also uninterpreted in the Darwin framework.

Darwin's support for architectural style is limited to the description of *parameterized configurations*. For example, a Darwin pipeline description is shown in figure 2.1. This description indicates that a pipeline is a linear sequence of filters, where each filter's output is bound to the input of the next filter in line.[2]

Using parameterization to describe families of systems means that only systems that can be described *constructively* can be effectively characterized. That is, in order to delineate membership in an architectural style it is essentially necessary to construct an algorithm that can construct exactly those members of the style. In our approach, a style's membership is explicitly defined via the properties that are shared by all members.

Darwin's support for architectural styles is further limited by its weak notion of connectors. The provides/requires model of connection enforces an *asymmetrical* model of interaction. It implies that there is always one party in an interaction that is responsible for definition of the protocol. Thus, interaction patterns can not be described independent of the component that provides it. This is a serious weakness for styles, since it is common to introduce and support new models of interaction in a style while leaving the definition of components to architects at the configuration level.

[2]Darwin does not define the semantics of its bindings however, so it is not clear what kind of interaction occurs between filters. That is assumed to be defined by the filter that **provides** the interface point and the underlying implementation platform.

Darwin is also limited by its lack of an explicit means of introducing new service types. Darwin assumes that the collection of service types is supplied by the platform on which the implementation is to be developed, and relies on the existence of service type names that are used, without interpretation, for compatibility checking. Thus, it can only provide relatively restricted consistency checks on configurations.

In the next chapter we will show how our approach, which provides an explicit means of defining the semantics of new connector types, supports a general form of consistency checking.

2.4.2 UniCon

UniCon [SDK+95] is an Architecture Description Language developed by Shaw *et al.* UniCon provides a configuration tool for constructing executable configurations based on component types and implementations and "connection experts" that support particular connector types. UniCon is similar to Darwin in that it provides tools for constructing executable configurations based on black-box implementations and that it has a fixed collection of interaction types, but it differs in its model of connectors.

Where Darwin provides interactions via asymmetrical provide/require declarations of implicit connectors, UniCon supports explicit, symmetrical and asymmetrical connectors. That is, an architectural configuration contains connector declarations that logically define an interaction. Each connector has a collection of *roles* that define what participants are expected in a given interaction. Component interfaces, rather than providing or requiring services, are defined by *players* which have a type (indicating the nature of the interaction expected) and a set of properties (providing details of the component's interaction at that interface). At the configuration step, players on components are associated with roles on connectors.

While UniCon's model of explicit connectors seems to provide promise for architectural style, by creating a place where new interactions might be defined and compositional rules elaborated, UniCon currently provides limited mechanisms for defining new connector types. New types can only be added by hand-implementing new connection experts. This adds to the collection of built-in, atomic types. Further, UniCon provides no means of describing or delineating families of systems. Thus, analysis of architectures in UniCon is limited to those tools supplied with the given connector types (which depend on specific implementations of components and connectors) and there is no way to describe architectural styles.

However, UniCon's model of architectural description is fully compatible with ours, and provides tools for implementation of systems that we do not. Thus, as we discuss in chapter 10, if we describe the semantics of UniCon's connector and component types in our notation, the advantages of both can be gained. Our abstract formal model provides analysis and consistency checking of configurations and styles, while UniCon provides tools for construction of executable systems within the architectural framework.

2.4.3 Rapide

Rapide [LAK+95] was developed by Luckham *et al.* It is an Architectural Description Language based on modelling computations and interactions as partially ordered event sets (or "posets") [Pra86]. Unlike Darwin and UniCon, which define the behavioral semantics of components via their implementations, Rapide defines component types (called "interfaces") in terms of a collection of communication events, which are either observed (Rapide calls these "extern actions") or initiated ("public actions"). Rapide interfaces define the computational behavior of a component by relating the observation of extern actions to the initiation of public actions.

Interaction between components is described for a particular configuration in one of two ways: First, events can be "connected," in which case they are aliased to the same event. Second, an architecture can declare "constraints," which specify causal relationships between events on different components. Given a constraint, the initiation of one event will result in the generation of another event following it in all event orderings, but they are not considered the same event.

Rapide's connections are similar to Darwin bindings in that they define an asymmetrical, primitive relation between two components. The type language of actions permits the definition, essentially, of function calls between components. (Events have parameters and possibly return values.) As such, they, like Darwin bindings, are not adequate for the introduction of new interaction types and do not support symmetric interaction patterns.

Rapide's constraints might appear to provide better support for an explicit connector mechanism, since related constraints can generate quite complex interactions between components. However, these constraints can only be declared at the global configuration level and therefore do not permit the localization of analysis of interactions. Further, because the language does not permit them to be explicitly bundled as connectors, complex interaction patterns (sets of constraints) can not effectively be reused in multiple contexts. As we will show, the ability to localize and reuse both interaction descriptions and analysis is critical to the practical characterization of architectural styles.

Rapide permits a form of consistency checking and analysis through *architectural simulation*. In essence, the architecture is simulated, generating a partially ordered set of events that is compatible with the interface, behavior, and constraint specifications. Because the generated set explicitly defines causal relations between events rather than simply providing one possible sequence of the events, simulation is useful for detecting execution alternatives such as race conditions.

However, a given simulation execution will provide only *one* possible poset, rather than all permitted posets. This means that alternatives due to non-determinism in a behavior specification are not captured by architectural simulation.

Suppose, for example, that a component is specified as in figure 2.2. This component has two different possible responses to the `A` event: either `B` and `C` will be generated (but be independent), or `C` and `D` will be generated (again, independently). This results in two different possible posets, shown in figure 2.3. On any given simulation execution, only *one* of these posets will be generated. This means that using simulation to check for architectural consistency is essentially *testing* the architecture, rather than an exhaustive analysis. Failure to locate a problem provides no guarantee that the inconsistency will not

```
type RapideComp is interface
   extern action A();
   public
      action B();
      action C();
      action D();
   constraint match
        A() -> ((B() and C()) or (C() and D()));
end RapideComp;
```

Figure 2.2: A non-deterministic Rapide component.

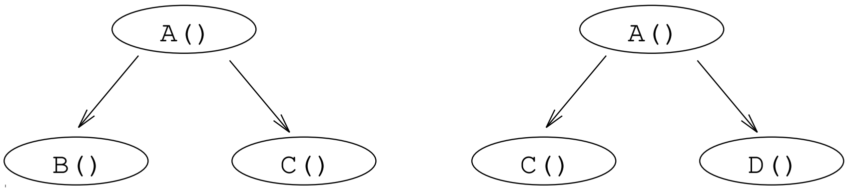

Figure 2.3: Two alternative posets.

arise on a different execution.

In our approach, we define consistency checks that can be applied *statically* to an architecture, rather than through simulation. Thus, our checks provide a stronger assurance of consistency than Rapide's. On the other hand, Rapide also provides a toolkit for instrumenting system implementations and then testing its conformance to the architectural specification by simulating the architecture in parallel with the execution of the implementation. Further, Rapide supports the description and simulation of dynamic reconfigurations.

2.4.4 Aesop

The Aesop family of architectural design environments [GAO94] was developed by Garlan *et al.* Aesop provides a vocabulary for architectural description through an object-oriented framework of types. An architectural configuration is represented as an interconnected collection of object instances, while the vocabulary of an architectural style is described by defining subtypes of the basic architectural types: *Component*, *Connector*, *Port*, *Role*, *Configuration*, and *Binding*. By default, an architectural configuration in Aesop is an unannotated hierarchical structure of components, connectors, and configurations. A style provides attributes for representing the semantics of individual elements, tools for analyzing and exploiting those specialized representations, and modifies the manipulation methods of the base types to enforce style-specific constraints.

For example, a pipe-filter style would define a "filter" subtype of component, a "pipe" subtype of connector, and appropriate port and role subtypes. The "insert port" method of a filter, for example, would require that the inserted port be either a data input or output. Additionally, tools could be introduced into the environment to generate implementations of pipe-filter systems from the architectural representation.

Unlike the other ADLs we have discussed, Aesop does provide an explicit mechanism for describing styles. However, Aesop, like much of the formal work, assumes that the semantics of an architecture may be arbitrarily different for any style. As a result, it provides no built in support for the description of a style's or a configuration's semantics. Rather, it is left to the tools and the subtypes' method definitions to provide this information *implicitly*. Thus, Aesop can be used to support the analysis techniques of a particular formal method or some other ADL (such as ours), but it provides no direct support.

2.4.5 ACME

ACME [GMW97] is another Architecture Description Language. Developed as a joint effort of several architectural research groups, ACME is intended to serve as a least-common-denominator interchange language for architectural descriptions. Like Aesop, ACME defines a basic, un-interpreted vocabulary of components, connectors, ports, roles, bindings, and configurations. These elements may have specifications associated with them through property lists. In ACME properties may be specified in any language. The semantics of the properties and, even, of the overall architectural specification are supplied by these auxiliary languages. The goal is that a specification written in one language, say UniCon, could share a common architectural structure with a specification in another language, say Rapide, and thus the architect will be able to take advantage of the descriptive and analytic tools of multiple Architecture Description Languages or formalisms. While the languages we have previously discussed focus on providing rich notations and tools for the kinds of architectural description and models that they support, ACME has focussed on providing a common skeleton through which the benefits of other languages and tools can be combined.

Thus, ACME is consistent with the ability to describe and analyze software architecture, but it does not, in itself, provide a sufficient basis. It is only through mappings to other languages that ACME descriptions can be interpreted and analyzed. As we discuss in chapter 10, a mapping between WRIGHT and ACME is currently being developed.

2.5 Programming Languages, MILs, and IDLs

2.5.1 Programming Language Module Facilities

A number of programming languages have facilities for describing the composition of systems at the module level. Examples include Ada packages [DoD83], Modula-3 modules [CDG$^+$89] and Standard ML structures and functors [HMM86]. These provide a means of developing components semi-independently of each other and composing them based on their interfaces.

Ada packages provide a means of limiting the visibility of the definitions of a program and for structuring the naming of constructs (such as types and procedures) in terms of the module structure. By creating a package with separate specification and body, the specification provides a boundary of abstraction and of information hiding, because the language enforces the restriction that there may be no dependency by one package of any information that is contained in another package and not exported via its specification. This facilitates the structuring of programs by limiting the inter-dependence of components and by reducing the extent to which name-conflicts will result in the inability to reuse a component in a particular context. However, the package facility has three important limitations that make it insufficient for use as an architecture description language.

First, the structure of a program is not separated from the components, but rather distributed throughout the packages. If one package uses, or depends on, another package, it refers to that other package explicitly, by name. The overall structure of the system can not be separated from the packages that make it up. If a package is to be used in multiple systems, then either all of the packages that it uses must be brought along, or the package must be modified to refer to the components in the new system.

Second, packages provide a means of limiting visibility of implementation constructs, not of supplementing those constructs with abstract characterizations. In particular, packages do not provide a means of capturing abstract patterns of interactions between packages. They define static interfaces and dependencies, but do not provide a basis for analysis of interactions.

Third, packages provide no means of explicitly capturing a family of systems. While it is obvious that in one sense, any collection of type definitions define a family of systems, (namely, those that use the types legally), this is a limited way of capturing the essential commonality between systems in an architectural style. An architectural style captures more than simply any legal configuration in a particular vocabulary. For example, in chapter 5 we describe a style which captures a class of distributed message-passing systems. An important problem in these systems is the possibility of system deadlock, in which the system ceases to make progress because every component is waiting for an action by some other component in the system. Our analysis of the style reveals that there are particular topologies of components that will deadlock even though they are are type-safe and carry out a well-defined computation. Thus, we define an explicit constraint on the use of the style's vocabulary that ensures that systems will not deadlock. This constraint is not possible to express using the Ada package facility, but it is desirable to provide this kind of explicit analysis and guidance to designers of this class of systems.

The functors of Standard ML [HMM86] provide a means of increasing module's independence, removing explicit references to other modules (termed structures in ML) via parameterization. A functor is essentially a structure with parameters. Parameters are supplied when the structure is to be instantiated, and hence reuse is increased by permitting a functor to be used in more contexts than a fully elaborated structure. This increase in flexibility overcomes some of the limitations of Ada packages, by providing a means of localizing composition information so that the topology of a configuration is described in a single structure that instantiates components, specified as functors.

However, ML, like Ada, provides no means of specifying the dynamic behavior of interactions between components independent of those components' implementations, or of specifying a family of systems that share a particular property.

2.5.2 MILs

A number of Module Interconnection Languages (MILs) have been developed to support the description of large-scale program structure independent of any particular programming language or system. These include languages such as MIL75 [DK76] and INTERCOL [Tic79], and are surveyed by Prieto-Diaz and Neighbors [PDN86]. These languages define program structure through definition/use bindings that associate definitions of program constructs such as data structures and functions with uses of those constructs. Like the module facilities of programming languages discussed above, these languages increase the independence of system components by creating visibility boundaries around modules. In addition, MILs provide an explicit and separable description of a system's structure by explicitly binding definitions to uses. However, also like current programming languages, these languages do not provide a means of specifying abstract patterns of interaction or of delineating families of systems. It is interesting to note that in their survey, Prieto-Diaz and Neighbors describe the limitations of MILs as follows:

> MILs are *not* concerned with what the system does (specification information), *how* the major parts of the system are embedded into the organization (analysis information), or how the individual modules implement their function (detailed design information). [PDN86, p. 307]

In this dissertation, on the other hand, we *are* concerned with the problem of architectural specification and analysis. In order to provide these capabilities, we must go beyond the structure-only framework of MILs to capture the behavior and interactions of software architectures.

2.5.3 MIFs and IDLs

2.5.3.1 MIFs

A number of systems have been developed to support the composition of software modules based on mechanisms other than definition/use bindings. One might call these Module Interconnection Formalisms (MIFs), because they extend the MILs with interconnection semantics. Examples of MIFs include Polylith [Pur94], FIELD [Rei90], and Durra [BWW88].

Each of these systems provides a particular interaction abstraction as the basis for system composition. Polylith provides run-time support for message passing between components based on a *software bus* abstraction. All communication is mediated by this bus, and thus data transformations and physical distribution are isolated from component implementation details. FIELD provides a software development tool integration model based on *event broadcast*. A tool registers a set of events (or messages) to which it can respond, and also broadcasts a set of events, without specifying the name of the recipient of that address. Communication is mediated by an event broadcast server that routes messages from senders to registered recipients. Durra provides a model of point-to-point buffered message passing. Interaction topologies, data-transformations, and clustering of components in threads are each described orthogonally and independent of component implementations.

In each case, these systems support the flexible implementation of systems that use the specific interaction mechanisms provided. However, they can not be used to build systems that use other communication patterns. Thus, they are not general mechanisms for describing interactions in software architectures.

2.5.3.2 IDLs

A common problem for the integration of systems is the potential for inconsistency in data representations, whether arising from the use of different programming languages or independent design and implementation. In order to address this issue, Interface Definition Languages (IDLs) have been developed to define abstract data interfaces and mappings from programming language constructs into intermediate representations that mediate the transformation between incompatible low-level formats. Examples of these languages include IDL [Lam87], UTS [HS87], and CORBA's IDL [OMG91].

Using these languages, incompatibilities between modules arising from data structure representations and differences in implementation platform can be resolved. Like MIFs, these systems define a single, restricted model of interaction, typically procedure call or object-method invocation. They define the data that is passed between components, but not the dynamics of the interactions. Rather, the dynamic constraints are implicit in the language mechanisms assumed to underly the IDL, or ignored, as when constraints *between* method calls on an object are not specified. Thus, like the MIFs, they do not provide a sufficient general basis for the description and analysis of interactions in software architectures.

2.5.4 Coordination Languages

Another approach to the combination of diverse computations is through coordination languages such as Linda [GC92]. These languages attempt to provide general mechanisms for the coordination of sequential components. Linda, for example, models an ensemble of computations as a bag of tuples, or a *tuple-space*, where a computation can read or consume tuples based on matching patterns in the tuple's data, and provides results by creating new tuples.

The premise of coordination languages is that the issues of coordination of components are orthogonal to those of sequential computation, and that by providing an explicit and separate language for coordination, generality and conceptual simplicity can be enhanced. The focus of coordination language research has been on showing how different kinds of coordination, such as fine-grain parallel computation, large-scale distributed systems, and uniprocessor task synchronization, can be cleanly and efficiently supported in a single model.

Our work similarly focusses on the description and analysis of system coordination independent of computations. However, we have taken a different approach, considering especially the *analysis* of interactions and the way in which *patterns* of interaction can be captured and exploited across multiple systems.

2.5.5 Wide-Spectrum Languages

While coordination languages seek to divide the development task, others have sought to provide greater levels of integration. Wide-spectrum languages such as REFINE [SKW83] provide a single notation supporting specification of systems at differing levels of abstraction, from very high-level specifications to executable, procedural representations.

To develop an executable program in REFINE, the developer begins by specifying the system using REFINE's high-level notations. This part of the language is similar to formal specification languages such as Z [Spi92] or VDM [Jon86]. This high-level description is then transformed through step-wise refinement, until it is expressed in the executable-sublanguage of REFINE. This is then compiled by traditional means, and executed.

The idea of this approach is to take advantage of explicit representations of the design knowledge (also represented in REFINE) to select and justify each transformation of the specification. In theory, development and maintenance is made simpler because the system's knowledge base provides suggestions about what transformations to apply and the use of explicit justifications increases both confidence in the resulting system and traceability from the original specification to the executable end-result.

The goal of a smooth development path, from high-level, non-implementation constraining specifications, through intermediate representations in which some but not all implementation decisions have been bound, to fully instantiated programs, is completely consistent with our approach to architectural design. Further, the idea of using explicitly represented design knowledge to guide the refinement by binding implementation decisions is fully consistent with our approach. Indeed, we believe that the effectiveness of architectural style in the design process is exactly due to the fact that a style represents the distillation of design knowledge: A style defines a particular solution to a problem together with constraints on when and how that solution can be applied.

In order to be effective, however, these wide-spectrum languages must include the ability to express systems at every different level of abstraction, and to represent all critical aspects of design, whether made explicitly by a developer or selected based on an automated knowledge-base, and to justify those decisions as valid refinement steps. Our work can thus contribute to the development of effective, usable wide-spectrum languages by providing a model for the representation of system structure and patterns of interaction (a critical design decision, needing to be represented and analyzed) and of architectural styles (a kind of design knowledge, representing a solution pattern to a problem, together with permitted contexts of use).

2.6 Object-Oriented Design

2.6.1 Object-Oriented Design Notations

A class of design notation that is related to software architectural description is that of object-oriented notations. Recently, several prominent object-oriented design notations [RBP+91, Boo93, Jac92] have been combined into a "Unified Modeling Language" [UML]. This notation provides a family of graphical notations for describing the attributes of and relations between objects in a design. These diagrams fall into two categories: Static structure diagrams and behavior diagrams. Static structure diagrams include class and object diagrams, while behavior diagrams include use case, sequence, collaboration, and activity diagrams.

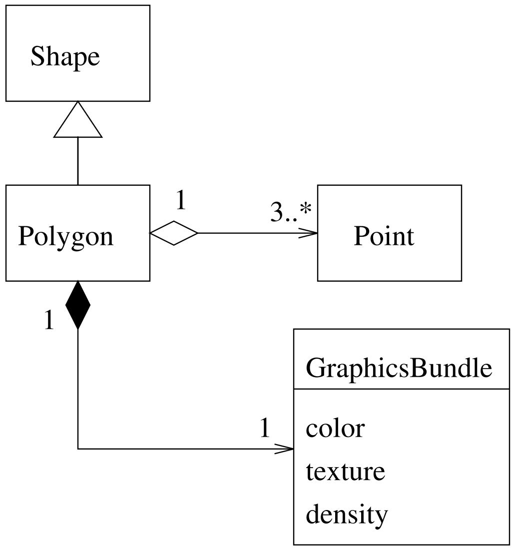

Figure 2.4: A UML static structure diagram.

Static structure diagrams capture the definitional and referential relationships between types, classes, and objects. Definitional relationships include objects which are instances of types or classes, classes which implement types, and types which are subtypes of other types. Referential relationships define objects that either composed from or contain references to other object instances. For example, figure 2.4 indicates that Polygon is a subtype of Shape and is composed of exactly one GraphicsBundle and three or more Points. A GraphicsBundle attribute has color, texture, and density attributes.

These diagrams do capture one kind of structure of a software system, namely, the dependencies between components in terms of their meaning (if one component is modified, the meaning of others is changed). However, it does not capture the behavior of the components or how they interact. Thus, they do not provide a sufficient basis for the description or analysis of patterns of interaction.

UML does provide diagrams intended to capture the behavior of a system. A collaboration diagram, for example, indicates sequences of communication actions between participants in a collaboration. Each box represents an object, while a line represents a pathway of communication. These lines are annotated with the names of messages or operations, with sequence numbers to indicate their ordering. For example, figure 2.5 shows the sequence of actions involved in redisplay of a collection of wires in a visual diagram. It indicates that when a user invokes redisplay() on a controller object, the following sequence of actions is the result: First, the `Controller` invokes `drawPositions` on the `wire` object. Next, `wire` invokes `drawSegment` on itself n times. For each of those invocations, `wire` calls `create` and then `display` on a new `Line` object. When a `Line` object receives `display`, it calls add on the `window` object, with itself (the `Line`) as a parameter.

These collaboration diagrams can be combined with state-machine representations of components based on Harel's Statecharts [Har87]. The state-machine diagrams provide additional structure to event orderings, providing the ability to indicate multiple possible

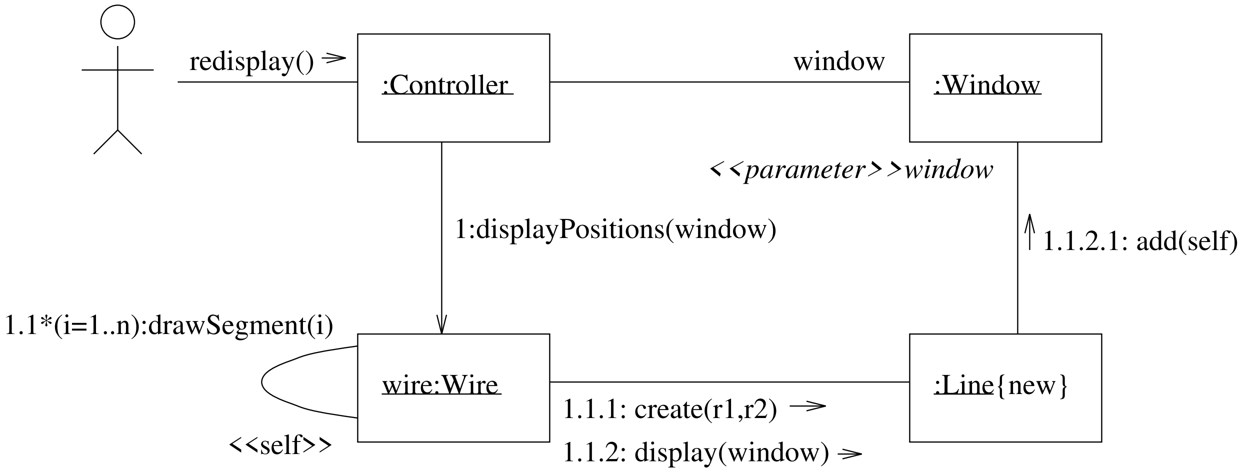

Figure 2.5: A UML collaboration diagram.

event sequences.

There are two important aspects of our approach to interaction patterns and architectures that these notations do not capture, however. First, the diagrams in UML do not distinguish between a pattern of interaction and the behavior of the participants in that interaction. Thus, it is difficult to reason about a protocol of interaction and a component's conformance to it, because the interaction is defined *in terms of* a given collection of components. If the components exactly match the overall behavior of the state machines, then they trivially obey the protocol. If they do not, then there is no clearly-defined way of separating out the parts of the components' behavior that are relevant to one particular interaction and analyzing that.

Second, the UML diagrams do not distinguish between descriptions of system configurations and families of systems. While there are constructs for describing conditions or alternatives within a single diagram, these mean that the system being described is free to bind these decisions at lower level of abstraction, not that there are multiple, different systems captured by the single diagram. When describing object-oriented *patterns*, object diagrams represent an *exemplar* of the pattern, rather than capturing the full variability inherent in a class of systems.

2.6.2 Extending Object-Oriented Models

In addition to the basic object-oriented notations described above, there has been research on extending object-oriented models in ways that relate to the description of architecture. For example, Nierstrasz [Nie93] extends object class definitions to include a finite-state process over the methods of the object. He uses these constraints relating the use of the methods of the object to define stronger subtyping relation and instantiation rules. While the motivation is similar to ours, Nierstrasz considers only one kind of component interaction: method invocation. This means that the semantics of his models, including the refinement relations that define subtyping and instantiation are specific to a single class of interaction.

Yellin [YS97] also defines a model of object composition based on protocols. He provides a restrictive definition of object compatibility and then increases the flexibility of the model using *software adaptors*. These adaptors have some of the flavor of architectural connectors, but are not fully general, since the purpose is to increase the flexibility of object composition—not to provide a separable and explicit definition of an interaction.

Recently, there has been an increasing interest in extending object-oriented design through *patterns* [GHJV95, BMR$^+$96, CS95]. A pattern captures design expertise by relating a specific problem in a particular context with a solution. The solution is commonly expressed as a framework of objects and informal text describing constraints and special semantic considerations.

Patterns are similar to software architectural styles in that they define a family of software systems that share common characteristics, but they differ in two important respects: First, a style specifically represents an *architectural* family, constructed of the architectural building blocks of components and connectors. Patterns span a range of levels of abstraction and parts of the software lifecycle from domain analysis, through software architecture, and down to the level of programming idioms. Second, the patterns community has focussed, to date, on the problem of how to tie solutions to problem contexts and how to select appropriate patterns, rather than on the description and analysis of prospective solutions. Thus, they have relied on informal, handbook-style, representations of system families, rather than on precise, semantically rich descriptions of systems. Also, any patterns of interaction are typically left implicit in the pattern of system composition, rather than called out as a separable concept. As we discuss in chapter 9, our techniques and notations can be applied to the solution part of patterns, thus enriching their precision and potential for analysis.

2.7 Other Support for Software Architecture

In addition to research efforts focussed specifically on description of software architecture, there have been efforts to address validation or evaluation of architectures. These models provide methods or criteria for the evaluation of architectures that are independent of particular formal notations or models. As such, they complement more formal approaches, but do not replace them. Each either assumes the existence of notations for describing architectures or lays out requirements for such notations without indicating how they are to be filled.

2.7.1 Architectural Review

AT&T Bell Labs have developed checklists and methods for the formal review of architectures in development organizations [ATT93]. The checklists capture issues that AT&T have found to be the most common, critical issues for systems that they develop. These issues include system integration, error recovery, administration and maintenance, and overall system performance. The methods define how these issues can be raised and addressed in the context of an entire development effort. In particular, they propose two formal architectural reviews, a "discovery" review to air issues that need to be addressed *before* an architecture is developed, and a "validation" review to discover problems with a proposed architecture before continuing to detailed design and implementation.

AT&T's review of development projects suggests that there are many issues, both functional and extra-functional, that must be addressed by a system in development. For our work, it is especially interesting to note the following item in their checklist:

> 10. Is the data being sent between systems in a consistent unit of information? (More than 50% of the trouble reports in some systems are related to communications interfaces within them.) [ATT93, p. 30]

In our vocabulary, this means that failure to properly define the connectors in a system's architecture is the source of more than half of the system's problems. This strongly supports our approach, by demonstrating the need for explicit description of connectors and their use in a configuration.

2.7.2 SAAM

Kazman *et al.* have developed a model, called SAAM, of architectural evaluation based on the development of scenarios [KBAW94]. Each of the scenarios is designed to capture what they term a "quality attribute," an extra-functional characteristic such as modifiability. The idea is that the term "modifiability" is too vague to be measured directly, but that a scenario can capture a specific aspect of modifiability, such as the ability to adapt to a new operating environment.

Like the AT&T work, Kazman *et al.* provide evaluation guidance through a method rather than a notation or model, relying on the informal expertize of architects both to describe the architecture to be evaluated and to determine how SAAM should be applied to it. They explicitly acknowledge that the validity of their techniques depend on the existence of a precise, well-defined architectural description. Thus, SAAM represents a technique that would benefit from the formalization of architectural description and that can be applied to formal descriptions.

2.7.3 Stakeholder Model

Gacek *et al.* have explored the extension of architectural models to better support the software lifecycle [$G^{+}95$]. They argue that an architectural description, describing the structure of a configuration, in isolation is insufficient to support evaluation of an architecture and the use of that architecture in development. For example, they note that, in addition to its structure, the rationale for an architecture must be captured so that later modifications do not violate key properties.

This view of architecture is consistent with our approach. While our particular architectural notation will only capture the architecture itself, and not the surrounding context such as requirements or rationale, it is certainly necessary to precisely define *what* the design is, as well as to capture other concerns. As we argue in chapter 10, both formal and informal approaches to architecture are necessary, and a formal architecture description language can serve as a centerpiece of a complete software process.

2.7.4 Aspect-Oriented Programming

Recently, Gregor Kiczales and others at Xerox PARC have begun to explore the possibilities of what they term "Aspect-Oriented Programming" [KLM$^+$97, LK97]. They argue that much of the complexity in programs results from the "tangling" of different aspects of the design of software systems. For example, control of synchronization and remote access issues cut across the functional aspects of a typical distributed system. They argue for the development of special-purpose "aspect languages" that separate the description of different issues from the "component language" appropriate for describing the functionality of a system. Different views of a system are then combined by an "Aspect Weaver" tool that generates combined code in a conventional programming language.

As we discuss in chapter 10, the decomposition of system description into views is an important part of architectural specification, and this work is a promising approach to understanding how to combine views effectively in an implementation. Currently, however, this work provides support only for specific views on a case-by-case basis, and will require further exploration before it can be applied, for example, to description of interaction protocols.

2.7.5 Law-Governed Architecture

Naftaly Minsky has proposed a model of system protocol enforcement based on his work in "law-governed systems" [Min91]. According to his model, a protocol is specified by a global collection of "laws," or constraints on how components may act depending on the actions that have occurred previously in the system. Each component is controlled by an enforcer, which ensures that no action is taken unless it is permitted by the law.

This mechanism provides support for the implementation of protocols in the context of open, evolving distributed systems. Because, in these systems, it is not possible to assume that all components have been explicitly and correctly implemented to obey a given protocol, enforcers are placed between untrusted components and the rest of the system.

This approach of runtime enforcement complements our own, specification-oriented approach. While we concentrate on the specification of protocols and checking whether components' specifications are consistent with a given protocol, Minsky's law-governed architecture represents an implementation mechanism by which the cost of a component diverging from its specification is limited. While a component can not be forced to do something that is expected of it, it can be prevented from doing something that it should not.

2.8 Possible Formal Bases

Our goal in this thesis is to provide a notation in which architectural patterns of asynchronous, discrete interaction and families of systems can be described and analyzed. There are a number of formalisms based on discrete actions, including state-machines, Petri nets, and process algebras, as well as tools for analysis of these systems. In this section, we discuss a selection of these models.

2.8.1 State Machines

A number of formal models have been proposed to extend basic finite-state machines to the description and analysis of interacting systems. Prominent among these is the Statecharts model, proposed by Harel [Har87]. This model extends basic finite-state automata to permit interactions between machines via shared actions. Transitions in the state graph are labelled with pairs of actions. The first action indicates an input to the automaton that is recognized and triggers the transition. The second action is an output of the automaton, that will result in other automata carrying out transitions. Thus, complex behaviors can be constructed by composing smaller state machines that synchronize on actions. Statecharts further provide hierarchical composition mechanisms, in which a "state" of one machine can actually be another state machine. In this case, transitions into the lower-level machine go to a designated start-state and transitions out of the lower-level machine are represented as transitions to a final state of the machine.

Another state-machine model is the I/O automaton model of Lynch and Tuttle [LT88]. I/O automata are similar to Statecharts. State transitions are triggered by input actions and may result in output actions. Like Statecharts, I/O automata are composed by matching the names of input and output actions.

Petri nets [Pet77] can be thought of as a variation on the state-machine model. Rather than having a single active state at a given point in its execution, a Petri net models the activation of states by having zero or more *tokens* present in the system. The multiple-token model is exploited by having transitions be from sets of states (the input states) to sets of states (the output states), rather than between single states. A transition is enabled when every input state has a token. The effect of a transition is to remove the tokens from each input state and to add a token to each output state.

Each of these models defines the executions of an automaton as a set of possible execution traces. Each trace represents a possible path through the state graph, alternating states (sets of states for Petri nets) and actions. These systems can be analyzed to ensure, for example, that there is always a transition possible for any input, and that selected undesirable states are not reachable.

For the purposes of modelling interaction patterns in architectures, however, the basic "sets of traces" model of execution omits a key property: The locus of decision in a system. It is common in a system to have a given trace where one of two different things will happen at a given point. If we are to analyze a pattern of interaction or to try to decide if a component conforms to that pattern, it is critical to understand where the decision is made about which of the alternatives will occur. It is important to distinguish between the case where a component is free to choose either alternative (or, say, only to provide one of the alternatives) and the case where the component is required to handle both alternatives. If we simply model the collection of possible executions, this can not be distinguished.

2.8.2 CSP

CSP [Hoa85] provides a different model of behavior from the state-machine models. Rather than describing a machine as a collection of states composed into a graph of transitions, a behavior is described through an algebraic model of processes, in which complex behaviors are constructed from simpler ones via a small set of operators. These operators include sequencing (one behavior occurs after another), alternative (one of two behaviors will occur), and interaction (two behaviors are combined by synchronizing on shared events).

For our purposes, an important property of CSP is its handling of choice. CSP provides two forms of choice: Internal choice, in which the object described by the process controls which behavior will occur, and external choice, in which the decision is made by the environment.

This distinction is key to the description of certain critical properties of architectural interactions. These properties include the ability to characterize the dynamic behavior of inter-component communication, to specify which components are responsible for making decisions during interaction, and to detect mismatched assumptions that could cause a component to get "stuck" midway through its interaction with another component.

FDR: Automated Analysis for CSP

Another piece of work related to our use of CSP is the development of FDR. One of the most successful forms of automated analysis for formal methods is exhaustive checking of a finite model [B^+90]. There are two kinds of exhaustive checker. One kind determines whether a given property holds in every state of a model, and the other determines whether two state machines are related in a particular way. Both kinds of tool exhaustively search the state space of a finite-state model and either conclusively determine that the check is satisfied or provide a counterexample (such as an execution of the state machine after which the property does not hold).

This technology is available for a variant of CSP through the tool FDR. FDR (which stands for "Failures, Divergence, Refinement") automates the test of whether one process refines another. A process P refines a process Q (written $\mathsf{Q} \sqsubseteq \mathsf{P}$) if the behaviors of P are consistent with (but possibly less general than) the behaviors of Q. The refinement test provides strong analytic capability for CSP in two ways: First, it provides a guarantee of substitutability of one process for another, much like subtyping between abstract datatypes. That is, if a process P refines a process Q, then, in any context in which Q appears, P can safely be substituted for Q.

Second, many properties can be expressed as an abstract "specification" process, such that any process that refines the specification process is guaranteed to have that property. This is because refinement permits a process to guarantee both the presence of a given behavior and the absence of others. If a given trace is not allowed, then the specification process omits that trace, and no process that includes the trace will be a legal refinement of it. If a given trace is required, then the specification process omits all other options from its refusals, and any process that refuses to engage in that trace will not be a refinement.

Chapter 3

Introduction to WRIGHT

3.1 Introduction

WRIGHT is designed to support the formal description of the architectural structure of software systems. In order to do so, it permits the description of both architectural styles, or families of systems, and architectural instances, or individual systems.

In this chapter, we present the elements of WRIGHT. The presentation will begin with a general discussion of the goals of an explicitly architectural description language. Then, we will show how WRIGHT meets these goals concretely. We start with the language of architectural types and instances and proceed to WRIGHT's use of architectural style. The intent of these initial sections is to convey a sense of how WRIGHT can be used.

Once all of the elements of the language have been described, we show in section 3.7 how tests can be applied to WRIGHT descriptions to determine both whether they are internally *consistent* and whether they are *complete*.

3.2 The Goals of an Architectural Description Language

Before describing WRIGHT it will be helpful to look at the general goals of any formal architectural description language. When an architect turns to a formal description language, he or she seeks two things: definition and analysis. In order to support definition of architecture, a notation must provide a precise semantics. Informal, ad-hoc notations such as box-and-line diagrams may be quite good at communicating the basic ideas and intuitions of the designer to others on the team. Trouble arises, however, when these intuitions must be translated into details. Then, the imprecision of the notation results in differing interpretations caused by unstated assumptions and omissions. This in turn means that inconsistencies can arise as the parts are refined towards implementation or as modifications are made. Thus, an architect or development team may turn to a formal description language in order to resolve ambiguity in the informal description, resulting in a shared understanding that can be used as a reference when questions come up.

Another goal of a formal description is to support analysis. In order to aid analysis, a notation must support reasoning about how the individual parts affect properties in an overall system and provide reassurances that the analysis made based on the abstract description will actually have validity in the implemented system. By describing a software system at a high level of abstraction, decomposing the computation into independent parts, the architect hopes to be able to understand how the whole system will behave and whether it meets the requirements that have been placed on it. A formal architecture description language hopes to aid in this goal by providing tools and techniques that can be applied to descriptions of the architecture. These tools and techniques give information about the system to be constructed based on that description. The description language should help not only to discover properties of the architecture, but also to realize the architecture as an implemented system.

To be useful, then, an architecture description language must at least provide two things: a precise semantics that resolves ambiguity and aids in the detection of inconsistencies, and a set of techniques that support reasoning about system properties. In an ideal setting, there would be automated tools available for common consistency checks and for generation of "routine" implementation details, while the notation would also support formal analysis of more complex or specialized properties.

This is not enough, however. In order to be usable, an ADL must fit the architect's own vocabulary. That is, the constructs in the language must match those that the architect uses in informal descriptions; if the notation requires too much effort to translate from the designer's intuition, then the designer will not find the notation worthwhile. It is important that the notation permit the designer to describe the abstractions that are important to the structuring of that system, and that the distinctions that matter to the architect can be made without laborious work-arounds.

Our notion of software architecture is based on the abstractions of the *component*, an independent computational entity, and the *connector*, a composition pattern among components. A software architecture language must expose these abstractions and provide means for the architect to use them in structuring the software system.

Because these abstractions are fundamental to the notion of software architecture, it is important that the language provide them directly, and not require that each architect build them from first principles for each new system. The abstractions of component and connector serve as the basic conceptual unit of an architecture; their absence can only obscure the very things that are most important to expose. To omit these constructs from the language is comparable to calling a programming language object-oriented but not providing data abstraction boundaries.

3.3 The Structure of WRIGHT

As an architectural description language, 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 component as computation and connector as pattern of interaction. In this section we describe the basic structural notations and show how they are combined to describe the software architecture of a system in a precise way. To emphasize these structural aspects of WRIGHT, we defer a discussion of specific notations for computation and interaction until section 3.4. In addition to simplifying the presentation, deferring the details of behavior specification also

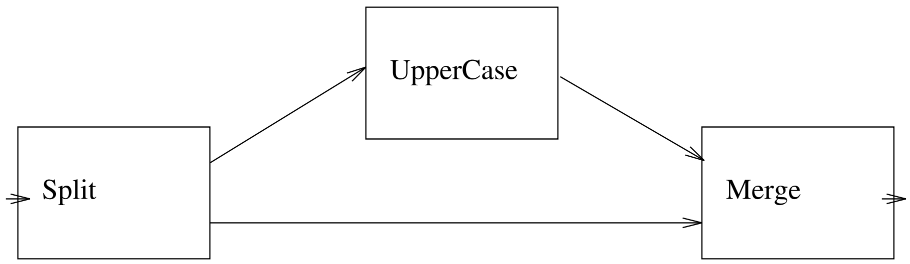

Figure 3.1: Box-Line drawing of example system.

reminds us that the structures are independent of the specific information attached to them; if there are critical issues not easily captured using WRIGHT's behavior specifications, the structuring provided by WRIGHT is still useful. This issue of alternative internal descriptions will be explored further in chapter 8.

Throughout the following introduction to WRIGHT, we use a simple example architecture to illustrate our ideas. This architecture, shown informally in figure 3.1, 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 do so, three components are used: SplitFilter, UpperCase, and Merge. In SplitFilter, the input stream is divided into two streams. The first stream is sent through UpperCase, which capitalizes every character, and the second is left unchanged. The two streams are recombined into the final output by Merge. Each of the streams is transmitted from one component to another via a pipe connector.

3.3.1 Components

A component describes a localized, independent computation. For example, in a pipe-and-filter system, a typical component might read all its input and convert each letter to upper case, or divide a single input stream into two, passing every other data element to a different output. Components in a database system might include a repository that provides access to its data and a client that summarizes the data in a report when it is requested by the user.

In WRIGHT, the description of a component has two important parts, the *interface* and the *computation*. An interface consists of a number of *ports*. Each port represents an interaction in which the component may participate. For example, a filter component might have three ports, one for input and two for output, as the SplitFilter in figure 3.2. 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 section of a description 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. The Split filter divides its input into two streams.

A port specification, which defines the interface of a component, indicates two aspects of a component. First, it indicates some aspect of the component's behavior. In this view, the port specification indicates the properties that the component must have if it is viewed through the lens of that particular port. The port becomes, in effect, a *partial specification* of the component. The **Computation** provides a more complete description of what is

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 3.2: The structure of a component description.

done.

The second aspect of a port is that it indicates the expectations of a component about the system with which it interacts. Although it is an informal specification, SplitFilter shows this aspect of a port as well. We can see that the SplitFilter *expects* to be able to read data on Input until it is notified of end-of-data. This aspect of a port specification will be clarified through our discussion of roles, attachments, and more precise behavior specifications.

In SplitFilter, notice how each port specification tells us something about the **Computation**. The Input port shows how data is read, and each of the Left and Right ports indicate part of the output behavior. Notice however that the port specifications do not combine to make a full specification of the computation. They are unable to relate the behaviors at different ports. There is no indication outside the **Computation**, for example, that Left and Right alternate. In WRIGHT, the **Computation** is itself the full specification, upon which analysis of the component's properties will be based; the purpose of the ports is to structure analysis of the component's interaction and to provide an additional level of abstraction, not to be redundant with the **Computation**.

This use of ports is similar to how abstract datatype declarations are used in a programming language. The run-time behavior of the type is completely determined by the body of the declaration; the method bodies and internal representation are sufficient for computations to be carried out. The abstract type declaration acts as an *interface* to the type, allowing consistency checks to be performed statically and to guide programmers' use of the type. An abstract datatype declaration provides a means of simplifying checks that the type will be used appropriately throughout the program. Similarly, we will see that the port and role declarations are needed for consistency checks even though they do not directly contribute to the system's meaning.

Note that, also like the elements of an abstract datatype's interface, the ports provide a means of referring to only part of the component (in the case of a port, part of the computation's behavior) rather than only discussing the whole component *en masse*, and each port refers to a different aspect of the component. Thus each port's name on a given component type must be unique within that component. This is also true of connector roles, introduced below.

3.3.2 Connectors

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. A procedure call is a simple kind of connector that indicates a call-and-return pattern of control. More complex connectors include database protocols such as two-phase commit and reliable, secure network message passing.

By making connector types explicit, WRIGHT seeks to achieve two important purposes: First, to extend the applicability of analysis, and second, to increase the independence of components. By drawing out a pattern of interaction in a connector type, and then using the pattern repeatedly in connector instances, WRIGHT makes explicit the commonality that typically occurs throughout a software architecture. The "pipe" interaction is used multiple times in a pipe-filter system, carrying data between different pairs of components. If each pair of components specifies the interaction independently, then there is no simple means of verifying that there is indeed only one kind of interaction throughout the system. If we analyze the pipe connector and demonstrate a property (that no data is lost, for example), then that property holds everywhere the pipe connector is used in the system. If we are forced to do the analysis in the context of a specific interaction, between the Left output of Split and the Input of UpperCase, for example, then we must repeat the analysis for every pair of components. The explicit description of connector types is 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, management of change and the structuring of implementation are simplified. If we use a runtime library, for example, to implement a communication pathway (such as `stdio` 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 lies ahead.

An important benefit of the use of explicit connectors is that it 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 that the component must meet, and an information-hiding boundary that clarifies what expectations the component can have about its environment. This means that a component specification is always explicitly described so that it can 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.

A WRIGHT description of a connector divides it into a set of *roles* and the *glue*. Each role specifies the behavior of a single 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 has a caller and a definer. An event broadcast connector has an announcer and zero or more listeners. A shared variable has one or more readers and writers.

Since a connector type is an abstract interaction, which will be used in different contexts with different components, there must be some indication of what the expectations are on any potential participants. Just as the ports of a component indicate how the component expects to interact in any connection, the roles indicate what is expected of any component that will participate in that interaction. Each role indicates what one of the components will do. In figure 3.3, for example, the Sink role indicates how that participant is expected

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 *receives data in same order delivered by* Source*]*

Figure 3.3: The structure of a connector description.

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 will not continue beyond end-of-data. Because the behavior specifications are currently informal, we don't know if the SplitFilter obeys the constraint that it must close the connection or not.

The **Glue** of a connector describes how the participants work together to create an interaction. In the case of a pipe (shown in figure 3.3), the glue describes how the data from the source is delivered to the sink. A procedure call would indicate that the caller initiates an invocation, followed by a return from the procedure definer. The connector **Glue** specifies how the computations of the components are composed to form a larger computation.

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

3.3.3 Configurations

In order to describe a complete system architecture, the components and connectors of a WRIGHT description must be combined into a *configuration*. A configuration is a collection of component instances combined via connectors.

Instances

Because there may be more than one use of a given component or connector in a system, we think of the descriptions described above as representing *types* of components and connectors. That is, they represent the properties of components, rather than actual examples of them in use. Thus "Pipe" is a type of connector, while there may be many instances of a pipe in a given system. In order to distinguish the different instances of each component and connector type that appear in a configuration, a WRIGHT description requires that each instance be explicitly and uniquely named. An example configuration with instance declarations is shown in figure 3.4. The same system is represented graphically in figure 3.1.

```
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 3.4: The structure of a configuration.

Attachments

Once the instances have been declared, a configuration is completed by describing the *attachments*. The attachments define the topology of the configuration, by showing which components participate in which interactions. This is done by associating a component's port with a connector's role. For example in figure 3.4, 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 3.4, 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 each of the elements of an architectural description. The component carries out a **Computation**, part of which is a particular interaction, specified by a **Port**. That port is attached to a **Role**, which indicates what rules the port must follow in order to be a legal participant in the interaction specified by the connector. If each of the components, as represented by their 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.

Hierarchy

WRIGHT supports hierarchical descriptions. In particular, the computation of a component (or the glue of a connector) can be represented either directly by a behavior specification (to be described in section 3.4) or by an architectural description itself. In the latter case, the component serves as abstraction boundary for a nested architectural subsystem.

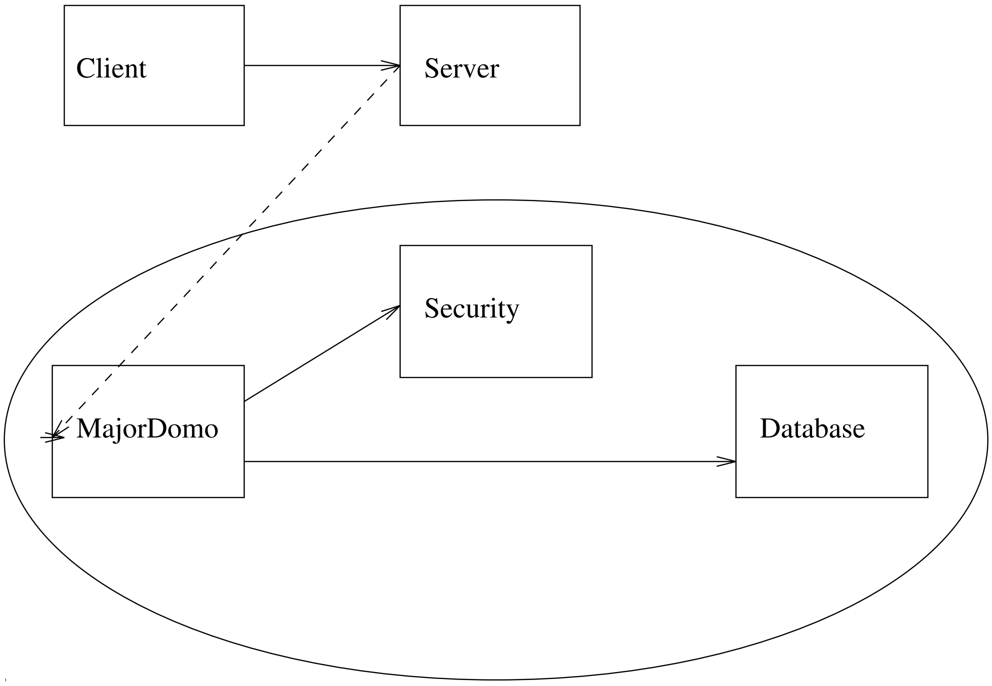

Figure 3.5: Hierarchical architecture.

An architectural subsystem 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 set of bindings, which define how the unattached port names on the "inside" are associated with the port names of the enclosing component. (Similarly for connectors: role names on the "inside" are identified with role names on the "outside".)

Figures 3.5 and 3.6 illustrate the use of hierarchy on a simple example. At the top level, the system consists of two components, a client and a server. The server is realized by a sub-architecture consisting of three components: A coordinator, that provides the service interface, a security manager, and a database.

This completes the structural portion of a WRIGHT instance description. It shows how the architectural abstractions of components and connectors are specified in WRIGHT. But we have yet to address the important questions of how we associate semantics with each of the elements, and how these semantic descriptions are combined, using the structuring notations, to model the system as a whole. We have also, so far, left open the question of when a WRIGHT description is legal: How can we detect inconsistencies or other problems with the architecture? These questions will be answered below, but first we complete our discussion of structure by extending these notions to the realm of architectural style.

3.3.4 Style

So far we have discussed how the structure of a single system can be defined: The architect introduces component and connector types and then uses these to create instances of components and connectors. Then, the topology of the configuration is specified through attachments of ports to roles.

```
Configuration HierServer
    Connector CSConn
      ...
    Component ClientType
      ...
    Component ServerType
      Port Service ...
      Computation
        Configuration SecureData
          Component Coordinator
          ...
        Instances
          C : Coordinator
          Security : SecurityManager
          ...
        Attachments
          C.Secure as S1.Client
          Security.Service as S1.Service
          ...
        End SecureData
        Bindings
          C.Combined = Service
        End Bindings
  Instances
    ...
  Attachments
    ...
end HierServer
```

Figure 3.6: Hierarchical specification in WRIGHT.

Often, however, an architect is concerned not with a single system in isolation, but rather with a system in the context of an entire family of systems. Before beginning a design, an architect commonly searches for an existing software system that has solved a problem similar to the one he is facing, or for tools and techniques that address the issues that are of particular concern. Also, an architect must address the probability that any software system will exist for a long time, requiring modifications to meet other challenges, thus, in effect, creating an entire family of similar systems.

Thus, an architect seeks not merely to develop an arbitrary architecture, but to select that architecture from a particular *style*, a family of systems. Using a style may guarantee that the architecture will have some set of desirable properties or allow the architect to use a vocabulary that is natural to the problem domain and supports reasoning and analysis about critical aspects of the system.

For example, if an architect is faced with a problem of transforming a sequential stream of input or a file in a regular way, the pipe-filter style 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 [Che88].

If, on the other hand, the system must maintain a complex data store and keep it up-to-date in a concurrent, potentially unreliable environment, the architect will instead look for a database-transaction style such as X/Open [XOp93]. This style of system construction permits the architect to make assumptions about the so-called ACID data properties: Atomicity, consistency, integrity, and durability. Other styles range from the domain-independent layered system style to very domain-specific styles for everything from robotic controllers to employee payroll systems.

In this section we show how WRIGHT can be used to define architectural styles. A style defines a set of properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations.

In WRIGHT, common vocabulary is introduced by declaring a set of component and connector types, using the declaration constructs introduced above for instance descriptions. The pipe-filter style, for example, would include a declaration of connector type Pipe. Then when a configuration is declared in the pipe-filter style, Pipes are automatically available for use.

3.3.4.1 Interface types

In addition to declaring complete component and connector types, a style's properties may constrain only part 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 stated, while the computation itself will differ between different filters. Also, the names and numbers of input and output ports will differ from filter to filter. the SplitFilter has one input and two outputs, while the UpperCase filter has one of each.

To describe these kinds of patterns and to simplify definition, a WRIGHT description can

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]*

Figure 3.7: Interface type declarations.

introduce *interface* types. For example the pipe-filter style would introduce DataInput and DataOutput interface types for use by filters. This new vocabulary is shown in figure 3.7. Interface types can then be used either as the port of a component (the most obvious use), or as the role of a connector. In the latter situation, the interface represents a constraint on the port interfaces that may be used in the role. Details of interface types will become clearer in section 3.4, when the specific behavior notations are introduced, and section 3.7.1.5, when the tests for attaching a port to a role are discussed.

3.3.4.2 Parameterization

In order to expand our descriptions from single systems to families of systems, we need to have the descriptions cover more situations. That is, they must be more flexible so that they can be used in more places in a description. In WRIGHT, we do this by *parameterizing* our type descriptions. We permit a description of a type to leave "holes" in the description that will be filled in when the type is instantiated.

For example, in the Unix pipe-filter style, all components have one input, named Stdin, and two outputs, named Stdout and Stderr. The interface to all Unix filters is the same. This is a pattern that we would like to describe in the Unix pipe-filter style. But the computation performed by each Unix filter is different. In order to describe this, 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 to be replaced with a *placeholder*, which is then filled with a parameter when the type is instantiated. So the type of a port or role, a computation, the name of an interface, *etc.* are all parameterizable. In addition, as we will see when we discuss formal behavior descriptions, *part* of a description of a port, role, computation, or glue can be a placeholder. These types are interpreted by substituting actual parameters for placeholders in the definition.

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 3.2, is an example of this kind of filter. But it has exactly two outputs. What if we want three, or four? In order for a style to explicitly list all of the possible Split filters with any number of outputs, we would need infinitely many descriptions. Instead, we make the number of outputs a parameter to the description:

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

A parameter accepting a range of integers is written "$\langle min\rangle..\langle max\rangle$." $\langle min\rangle$ indicates the smallest acceptable integer, and $\langle max\rangle$ indicates the largest. If one of the bounds is omitted (as is the case with $\langle max\rangle$ 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. In ranges appearing in port or role name subscripts, both minimum and maximum bounds must be defined. (We can't have an infinite number of ports, after all!)

Thus, the number of ports on a component is set at instantiation time and can not be changed during the execution of the system. This reflects the *static* nature of a WRIGHT description. As we will discuss in chapter 9, WRIGHT assumes that, at run time, the set of components and the interaction topology do not change. As we will elaborate, many kinds of "dynamic" architectures (where components appear and disappear, or where the topology of a configuration 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 instant.

3.3.4.3 Constraints

Clearly the new constructs of interface types and parameterization could equally well be used in the definition of a single configuration. If a system is large enough to contain repetitions, or if the architect wants to emphasize commonalities between different parts of a single architecture, these elements of the language will 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 of a pipeline 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, using the analytic techniques 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. If a system contains only pipes and filters, is this a coincidence because no other constructs turned out to be necessary, or is it an intrinsic property of the style that is used to simplify the implementation while retaining a high level of abstraction?

In order 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 WRIGHT definition of the pipe-filter style would indicate that all connectors must be pipes as follows:

$$\forall\, c : \mathit{Connectors} \bullet \mathit{Type}(c) = \mathsf{Pipe}$$

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.

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

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

Figure 3.8: The pipe-filter style.

$$\begin{aligned}
&\exists\, center : Components \bullet \\
&\quad \forall\, c : Connectors \bullet \exists\, r : Role;\ p : Port \mid ((center, p), (c, r)) \in Attachments \\
&\wedge\ \forall\, c : Components \bullet \exists\, cn : Connectors;\ r : Role;\ p : Port \\
&\qquad \mid ((c, p), (cn, r)) \in Attachments
\end{aligned}$$

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

The complete pipe-filter style specification is given in figure 3.8.

3.3.5 Extending Styles

An important way of defining new styles is as a *sub-style* of another style. We define a style to be a sub-style of another if it has all of the constraints of the other style. For example, we might define the pipeline style to consist of linear sequences of filters interacting via pipes. This style has all of the constraints of the pipe-filter style (all connectors are pipes and all components are filters), and so it is a sub-style of the pipe-filter style. We define sub-styles by giving the name of the super-style after the name of the sub-style. This has the effect of including all of the super-style's defined vocabulary (by inclusion) and constraints (by and-ing them with the sub-style's constraints). For example, we could define a pipeline as follows:

Style Pipeline : Pipe-Filter
Constraints

$$\begin{aligned}
&\#Components = \#Connectors + 1 \\
&\wedge\ \forall\, c : Components \bullet \exists\, p : Ports(c);\ cn : Connectors;\ r : Roles(cn) \bullet \\
&\qquad ((c, p), (cn, r)) \in Attachments \\
&\wedge\ \exists\, s\colon \mathrm{seq}\ Components \mid \mathrm{ran}\, s = Components \wedge \#s = \#Components \\
&\qquad \bullet \forall\, cn : Connectors;\ i, j : \mathbb{N};\ p_1 : Ports(s(i));\ p_2 : Ports(s(j)) \mid \\
&\qquad\quad \{((s(i), p_1), (cn, \mathsf{Sink})), ((s(j), p_2), (cn, \mathsf{Source}))\} \subseteq Attachments \\
&\qquad\quad \bullet\ i = j - 1
\end{aligned}$$

3.4 Specifying Behavior

So far, we have seen how to specify the structure of an architecture, and how an architectural style can be used to describe a family of similar structures. But our specifications of the meaning of the architectural types and of the behavior of the elements has been informal. What precisely do we mean when we 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 in an interaction is in control at any point? What restrictions are there 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. For historical reasons, a pattern of behavior is termed a *process* in CSP.

3.4.1 Events

The basic unit of a CSP behavior specification is an *event*. An event represents an important 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 a 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 within that process' definition: The specification of the DataOutput port would use the event $\overline{\text{write}}$ 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 $\overline{\text{end-of-data}}$. A special event in WRIGHT is $\surd$, which indicates the successful termination of the entire system. Because this event is not actually a communication event (it represents the ending of all further communication), 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: the source of data for a pipe supplies data when it writes to the pipe: $\overline{\text{write!x}}$. If a process receives data, this is input, and written with a question mark: e?x. Notice that output is usually signalled ($\overline{\text{e!x}}$) and input is usually observed (e?x).[1]

3.4.2 Processes

Given the basic element of behavior, the event, it is possible to construct patterns of behavior, or processes. Processes are described by combining events and other, simpler

[1]This is not always the case, however. Consider a component that reads data from a variable. While the component initiates this event, it receives 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).

processes. The simplest 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 $\mathsf{e}{\rightarrow}\mathsf{P}$ is the process that first engages in the event e and then behaves as P. For example, we define the *success process*, $\S$, to be $\surd{\rightarrow}$**STOP**, the process that successfully terminates immediately.

Another form of sequencing is the ";" operator. This combines two processes in sequence. $\mathsf{P} ; \mathsf{Q}$ is the process that behaves as P until P terminates successfully and then behaves as Q. For example $(\mathsf{e}{\rightarrow}\mathsf{f}{\rightarrow}\S) ; (\mathsf{g}{\rightarrow}\S) = \mathsf{e}{\rightarrow}\mathsf{f}{\rightarrow}\mathsf{g}{\rightarrow}\S$.[2] If the process P does not terminate, then $\mathsf{P} ; \mathsf{Q}$ acts as P forever.[3]

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 particular processes, we can describe behavior patterns that occur over and over. 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 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:

$$\mathsf{P}_V = \mathsf{Q}, \textbf{ when } p(V)$$

defines a process P over variables V only when the boolean expression $p(V)$ is true. Multiple alternatives are indicated by stacking them with a large curly-brace. For example:

$$\mathsf{P}_1 \textbf{ where } \mathsf{P}_i = \begin{cases} \overline{\mathsf{count!}i}{\rightarrow}\mathsf{P}_{i+1}, & \textbf{when } i < 3 \\ \overline{\mathsf{count!}i}{\rightarrow}\mathsf{P}_1, & \textbf{otherwise} \end{cases}$$

[2]We use the convention that $\rightarrow$ associates to the right: $\mathsf{e}{\rightarrow}\mathsf{f}{\rightarrow}\mathsf{P} = \mathsf{e}{\rightarrow}(\mathsf{f}{\rightarrow}\mathsf{P})$.

[3]Note that we use ";" in a slightly different fashion than it is often interpreted in CSP. See appendix A for a discussion.

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. We call this *deterministic* or *external choice* and use the operator $\Box$. The process $\mathsf{e} \rightarrow \mathsf{P} \Box \mathsf{f} \rightarrow \mathsf{Q}$ 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 form of choice is called deterministic because the behavior of the process is entirely determined by what the environment does.[4] 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 $\overline{\mathsf{e}} \rightarrow \mathsf{P} \sqcap \overline{\mathsf{f}} \rightarrow \mathsf{Q}$ is the process that will either output $\overline{\mathsf{e}}$ and then act as P or output $\overline{\mathsf{f}}$ 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.

To make processes more flexible, the sequencing and choice operators can also be quantified over a set: $\langle op \rangle x : S \bullet \mathsf{P}(x)$. 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,

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

If the sequencing operator ';' is used, "$; x : S \bullet P(x)$", we must take into account the fact that it is not symmetric ($\mathsf{P}; \mathsf{Q} \neq \mathsf{Q}; \mathsf{P}$). The meaning of quantification over sequence is some unspecified sequencing of the processes:

$$\begin{aligned}(; x : S \bullet P(x)) = (\sqcap\, x : S \bullet (P(x)\, ; \\ (; y : S \setminus \{x\} \bullet P(y)))).\end{aligned}$$

Thus,

$$\begin{aligned}(; i : \{1, 2, 3\} \bullet \mathsf{P}_i) = \quad & (\mathsf{P}_1; \mathsf{P}_2; \mathsf{P}_3) \sqcap (\mathsf{P}_1; \mathsf{P}_3; \mathsf{P}_2) \\ \sqcap\, & (\mathsf{P}_2; \mathsf{P}_3; \mathsf{P}_1) \sqcap (\mathsf{P}_2; \mathsf{P}_1; \mathsf{P}_3) \\ \sqcap\, & (\mathsf{P}_3; \mathsf{P}_1; \mathsf{P}_2) \sqcap (\mathsf{P}_3; \mathsf{P}_2; \mathsf{P}_1).\end{aligned}$$

Three forms of this operator are typically used:

- $\Box\, x : S \bullet \mathsf{P}(x)$ indicates an external choice between different $\mathsf{P}(x)$,
- $\sqcap\, x : S \bullet \mathsf{P}(x)$ indicates an internal choice between different $\mathsf{P}(x)$, and
- $; x : S \bullet \mathsf{P}(x)$ indicates the execution of all of the different $\mathsf{P}(x)$ in some order.

[4]Technically, the process is only deterministic when the guarding events (e and f in our example) are different.

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

3.4.3 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 = $\overline{\text{call}} \to \text{return} \to \text{Caller} \sqcap \S$
  Role Definer = $\text{call} \to \overline{\text{return}} \to \text{Definer} \; \square \; \S$
  Glue = $\text{Caller.call} \to \overline{\text{Definer.call}} \to \textbf{Glue}$
    $\square \; \text{Definer.return} \to \overline{\text{Caller.return}} \to \textbf{Glue}$
    $\square \; \S$

There are three elements of this definition worth noting. First, the Caller and the Definer use different alternatives 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 will occur or not.

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 the Caller component executing 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 *corresponds* 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. Where a role represents the behavior of a component, the **Glue** represents the composition of different components. Thus, the **Glue**'s use of initiate and observe 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, "$\text{Caller.call} \to \overline{\text{Definer.call}} \to \textbf{Glue}$" indicates that the Definer will observe a call event following its initiation by the Caller. "$\text{Definer.return} \to \overline{\text{Caller.return}} \to \textbf{Glue}$" indicates that the Caller will process a return following the signal by Definer.

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, as we discuss in chapter 9, provide "syntactic sugaring" to simplify glue descriptions for common cases.

[6]In practice, these events actually occurrence simultaneously in the software system, but for various technical reasons (discussed in chapter 9) WRIGHT considers all events in different roles to be distinguishable events.

Suppose we wanted to define a more complicated form of procedure call, in which there was more than one possible caller. In this case, the connector would need to specify how the different callers are coordinated with the definer so that the result from one caller's request is not sent to a different caller. One possible definition is as follows:

Connector MultiCallers(nc:1..)
  Role $\mathsf{Caller}_{1..nc} = \overline{\mathsf{call}} \rightarrow \mathsf{return} \rightarrow \mathsf{Caller} \sqcap \S$
  Role $\mathsf{Definer} = \mathsf{call} \rightarrow \overline{\mathsf{return}} \rightarrow \mathsf{Definer} \mathbin{\Box} \S$
  Glue $= (\Box i : 1..nc \bullet \mathsf{Caller}_i.\mathsf{call} \rightarrow \overline{\mathsf{Definer.call}}$
    $\rightarrow \mathsf{Definer.return} \rightarrow \overline{\mathsf{Caller}_i.\mathsf{return}} \rightarrow$ **Glue**)
    $\Box\, \S$

The definition of MultiCallers provides only *one* interpretation of how multiple callers can be coordinated. It indicates that all calls are handled in sequence, so that the definer is only dealing with one at a time. Alternative definitions might permit nested calls, or arbitrary interleaving of calls (in which case the definer would need to know about and supply handles identifying which result went with which request). Thus, we can not, in general, ignore the glue and provide a default interpretation.

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

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, indicated by the $\sqcap$ 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 (the writer may have signalled end-of-data). In this case, we *must* close. All of these situations are covered in the formal definition:

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

Initially, DataInput has an internal choice between the events $\overline{\mathsf{read}}$ and $\overline{\mathsf{close}}$. This indicates that the component may choose which of these to initiate. Once $\overline{\mathsf{read}}$ has been initiated, DataInput has an external choice between data?x and end-of-data. This indicates that a component must be able to accept either of these occurrences. The event end-of-data is followed immediately by $\overline{\mathsf{close}}$ without any choice operators, indicating that the component is required to initiate close whenever it observes end-of-data.

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:

$$\textbf{Glue} = \mathsf{Open}_{\langle\rangle}\ \textbf{where}$$

$$\mathsf{Open}_s = \begin{cases} \mathsf{Source.write?x} \rightarrow \mathsf{Open}_{\langle x \rangle} & \\ \Box\ \mathsf{Source.close} \rightarrow \mathsf{Closed}_{\langle\rangle} & \\ \Box\ \mathsf{Sink.close} \rightarrow Capped, & \textbf{when } s = \langle\rangle \\ & \\ \mathsf{Sink.read} \rightarrow \overline{\mathsf{Sink.data!x}} \rightarrow \mathsf{Open}_{s'} & \\ \Box\ \mathsf{Source.write?y} \rightarrow \mathsf{Open}_{\langle y \rangle ^\frown s' ^\frown \langle x \rangle} & \\ \Box\ \mathsf{Source.close} \rightarrow \mathsf{Closed}_{s' ^\frown \langle x \rangle} & \\ \Box\ \mathsf{Sink.close} \rightarrow \mathsf{Capped}, & \textbf{when } s = s' ^\frown \langle x \rangle \end{cases}$$

$$\mathsf{Closed}_s = \begin{cases} \mathsf{Sink.read} \rightarrow \overline{\mathsf{Sink.data!x}} \rightarrow \mathsf{Closed}_{s'} & \\ \Box\ \mathsf{Sink.close} \rightarrow \S, & \textbf{when } s = s' ^\frown \langle x \rangle \\ & \\ \mathsf{Sink.read} \rightarrow \overline{\mathsf{Sink.end\text{-}of\text{-}data}} \rightarrow \mathsf{Sink.close} \rightarrow \S & \\ \Box\ \mathsf{Sink.close} \rightarrow \S, & \textbf{when } s = \langle\rangle \end{cases}$$

$$\begin{aligned} \mathsf{Capped} = {} & \mathsf{Source.write?x} \rightarrow \mathsf{Capped} \\ & \Box\ \mathsf{Source.close} \rightarrow \S \end{aligned}$$

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 $\mathsf{Closed}_{\langle\rangle}$. In this situation, the Sink will be informed of end-of-data, and then only close is possible. In the state Capped, the Sink has closed before the Source. In this case, there is no need to remember any data values, but the Source is free to continue until it closes of its own accord.

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**

$$\text{Transfer}_i = \begin{cases} \overline{\text{Input.read}} \rightarrow & (\text{Input.data?x} \rightarrow \overline{\text{Output}_i\text{.write!x}} \rightarrow \text{Transfer}_{i+1} \\ & \;\square\; \text{Input.end-of-data} \rightarrow \text{Close}), \\ & \qquad\qquad \textbf{when } 1 \leq i \wedge i \leq nout \\ \text{Transfer}_1, & \qquad\qquad \textbf{when } i = nout + 1 \end{cases}$$

$$\text{Close} = \overline{\text{Input.close}} \rightarrow (;\ i : 1..nout \bullet \overline{\text{Output}_i\text{.close}} \rightarrow \S)$$

3.5 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: What do the behavior patterns of the different elements imply about the system as a whole? and, 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. Details of the underlying CSP model that we use are given in appendix A.

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, complete behavior of the components and connectors, respectively. In this section we explain how these distinct processes work together to define the behavior of the configuration and help us to determine whether the configuration contains inconsistencies that mean the system cannot operate correctly.

3.5.1 Parallel Composition

Abstractly, we think of the behavior of an architectural configuration as consisting of each 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. 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:[7]

$\mathsf{P} = (\mathsf{e} \rightarrow \mathsf{f} \rightarrow \mathsf{P}) \,\square\, (\mathsf{g} \rightarrow \mathsf{P})$, and
$\mathsf{Q} = \mathsf{e} \rightarrow (\mathsf{f} \rightarrow \mathsf{Q} \,\square\, \mathsf{h} \rightarrow \mathsf{Q})$.

What will happen if we combine these in parallel, as $\mathsf{P}\|\mathsf{Q}$? 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 $\langle \mathsf{e}, \mathsf{f} \rangle$ sequence, both P and Q are in their original states, so the sequence repeats. Thus, the process $\mathsf{P}\|\mathsf{Q}$ is equivalent to the process $\mathsf{R} = \mathsf{e} \rightarrow \mathsf{f} \rightarrow \mathsf{R}$.[8]

3.5.2 Alphabets

Sometimes, we use a process to describe only some of the events that may occur in a larger computation. This process does not make any reference to some of the possible events and should not therefore control whether they occur or not. For example, if we have a system with two filters in a pipeline (see figure 3.9), 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 3.9.[9]

$\mathsf{A} = \mathsf{a} \rightarrow \mathsf{A} \sqcap \S$
$\mathsf{C} = \mathsf{a} \rightarrow \mathsf{c} \rightarrow \mathsf{C} \,\square\, \S$
$\mathsf{B} = \mathsf{c} \rightarrow \mathsf{b} \rightarrow \mathsf{B} \,\square\, \S$

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 sequence $\langle \mathsf{c}, \mathsf{b} \rangle$ 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: $\mathsf{A}\|\mathsf{C}\|\mathsf{B}$?

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 this example, we assume that the alphabets of all processes are identical.

[8] 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 appendix A.

[9] 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.

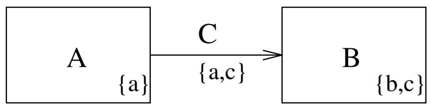

Figure 3.9: A and B should interact *only* via C.

To address this issue, CSP uses the concept of an *alphabet*. The alphabet of a process (written $\alpha\mathsf{P}$) is the set of events over which it has an influence. In a parallel composition, a process controls the events that are in its alphabet and ignores the others. In our example, the alphabet of A is the set $\{\mathsf{a}\}$, the alphabet of B is $\{\mathsf{b}, \mathsf{c}\}$, and the alphabet of C is $\{\mathsf{a}, \mathsf{c}\}$. Thus, in the composition $\mathsf{A}\|\mathsf{C}\|\mathsf{B}$, 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 consider all of the influences on the part we are interested in? In some programming languages there is no easy way to avoid looking at the code of the entire system. If we use techniques such as data abstraction or have a software architecture that is specified and followed, then there are clear scoping boundaries that we can use to limit the potential effect on a local element. Note, however, that unlike scoping in programming languages, we define the scope of an event to be an interaction. We do not require that scoping be hierarchical, where it is either visible to all or no components at a given level.

3.5.3 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 the component **Computations** interact only according to the constraints of the connectors' **Glue**. 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, just as we expect from section 3.3.

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 main difficulties here:

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 3.4) 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 the objects 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 an other in a program, either the two libraries must be forced to use exactly the same set of procedure names, or the program must be changed to incorporate the new library. Either solution can be costly or even 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→b→P, the result is a process with the same structure, but with different event names: *shift*(P) = b→c→*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*. They are written L : P to indicate the process P with its events prefixed by the label L. Thus, an instance of the SplitFilter named Splitter would refer to its events with the name Splitter: Splitter.Left.write, Splitter.Left.close, *etc.* This way, there can be multiple instances of a type and they will not interfere.

Relabelling is sufficient to construct processes to represent each component instance.[10] 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).

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.

[10] We will, however, require the full power of renaming when we discuss attachments, below.

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

Definition 1 *For any names* N,N',M, M', *not necessarily distinct,*

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

In the case of the attachment above, we would thus use $\mathcal{R}^{(\mathsf{P_1},\mathsf{Source})}_{(\mathsf{Splitter},\mathsf{Left})}$. 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.

3.5.4 Configuration

How does this all work out in practice? Let's look at a simple example. Suppose we want to make the three processes A, B, and C from above into actual WRIGHT components and connectors. Then they would look something like this:

Configuration ABC
    Component A-type
        Port Out = $\overline{\text{a}}$→Out ⊓ §
        Computation = $\overline{\text{Out.a}}$→**Computation** ⊓ §
    Component B-type
        Port In = c→In [] §
        Computation = In.c→$\overline{\text{b}}$→**Computation** [] §

    Connector C-type
        Role Origin = $\overline{\text{a}}$→Origin ⊓ §
        Role Target = c→Target [] §
        Glue = Origin.a→$\overline{\text{Target.c}}$→**Glue** [] §

  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:[11]

[11]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.

$\mathsf{A} = \overline{\mathsf{A.Out.a}} \rightarrow \mathsf{A} \sqcap \S$
$\parallel \mathsf{C} = \mathsf{A.Out.a} \rightarrow \overline{\mathsf{B.In.c}} \rightarrow \mathsf{C} \; [] \; \S$
$\parallel \mathsf{B} = \mathsf{B.In.c} \rightarrow \overline{\mathsf{B.b}} \rightarrow \mathsf{B} \; [] \; \S$

Formally, we define a configuration behavior as follows:

Definition 2 (Configuration Behavior) *If a configuration declares component instances* Cp_1:CpT_1 ... Cp_n:CpT_n*, where each component type* CpT_i *has computation process* CpP_i*, connector instances* Cn_1:CnT_1 ... Cn_m: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* $(\parallel i : 1..n \bullet \mathsf{Cp}_i : \mathsf{CpP}_i) \parallel (\parallel j : 1..m \bullet \overline{\mathcal{R}}(\mathsf{Cn}_j : \mathsf{CnP}_j))$.

In this definition, it is worth noting the definition $\overline{\mathcal{R}} = \mathcal{R}_1 \circ ... \circ \mathcal{R}_k$. This indicates that the attachment functions are composed to form a single function. Recall that the definition of the attachment functions made it a *total* function over events, but that only the relevant events (of the specific role on the connector) are changed by the definition. The requirement that all connector names be unique and all roles be attached to at most one port ensures that there will be no conflicts when composing attachment functions in a configuration.

Hierarchy

As we discussed previously, an important way of describing the computation of a component is by *hierarchical decomposition*. A component is decomposed by describing it as a full configuration. In order to achieve this, it is necessary to provide *bindings* that define the correspondence between parts of the configuration and the ports of the component. Thus, it is shown how the configuration interacts with its environment. The technique of using bindings ensures that the component truly encapsulates the architectural structure, because there is no way for any user of the component type to tell whether there is an atomic **Computation** specified or whether there is a configuration.

In this section we show how a configuration, in conjunction with bindings, can be used as a component type.

According to definition 2, the WRIGHT interpretation of a configuration is a CSP process which includes a process for each component and each connector. In the behavior process, each event that is mentioned in a port is given a three-level name of the form C.P.e, where C is the name of the component instance and P is the name of the port on that component. We used renaming functions to map the connector glue events into this same naming structure.

A component type that uses an event in its port gives the event a two-level name, using the name of the port. Thus, in a component type, events are of the form P.e, rather than C.P.e. We must thus once again rename the events so that they are of this two-level form. The bindings guide us in this.

We must also ensure that any events that are not used in a port do not interfere with interpreting the behavior at that port later. For example, recall that in our example of section 3.5.3 we had a configuration with two components A and B. B had an internal event b, that when we used it in an instance, was renamed to have a two level structure B.b. This was not a problem, because in a component instance, every event on a port has a *three* level structure, for example B.In.c. But in a component type (as we now intend to use the configuration), port events once again have a two level structure. What if the component type has a port named B? Then the event B.b might be misinterpreted as belonging to that port. Thus, we will want to expose only events that are properly renamed via bindings. This projection will occur before renaming so that we avoid capturing the events.

The relabelling is simple enough to accomplish using the $\mathcal{R}$ renaming function. Instead of using a Connector.Role to Component.Port mapping, we simply use Component.Port to NewPort: Each binding "C.P **to** P'" has a corresponding renaming function $\mathcal{R}^{\mathsf{C.P}}_{\mathsf{P'}}$. The binding also defines a *keeping event set* C.P:Σ which is all events of the form P'.*e*. These are the events that are exposed at the port of the outer component. All others will be hidden when the configuration is enclosed hierarchically.

These are combined to make a complete transformation as follows:

Definition 3 (Hierarchy) *If a component* C *declares its computation as a configuration with behavior process* Conf *and a set of bindings with renaming functions* $\mathcal{R}_1..\mathcal{R}_n$ *and keeping event sets* $S_1..S_n$*, let* $\overline{\mathcal{R}} = \mathcal{R}_1 \circ ... \circ \mathcal{R}_n$ *and* $\overline{\mathcal{S}} = S_1 \cup ... \cup S_n$*. Then its computation process is* $\overline{\mathcal{R}}(\mathsf{Conf} \upharpoonright \overline{\mathcal{S}})$.

3.6 Semantic Style Constraints

In section 3.3.4 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 have so far discussed are *syntactic*: They refer to the topology of instances and attachments or constrain 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. However, 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:

$$\begin{array}{l}\forall\, \mathsf{C} : Components;\ \mathsf{e} : \alpha_o\mathsf{C};\ t : Traces(\mathsf{C}) \mid \mathsf{e} = last(t) \\ \quad \bullet \{\mathsf{f} : \alpha\mathsf{C} \mid t \frown \langle f \rangle \in Traces(C)\} = \{\overline{\mathsf{Log.log!e}}\}\end{array}$$

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.[12]

[12]The freedom to apply a constraint to components with varying alphabets highlights the fact that these

In order to support these semantic constraints, we add the following predicates to the constraint language described in section 3.3.4. In each case, we use an element (component, connector, port, or role) and its behavior process interchangeably. Constraints refer to processes *before renaming*.

- αP: the alphabet of process P.
- α_iP: the subset of αP that is initiated.
- α_oP: the subset of αP that is observed.
- *Traces*(P): the traces of process P.
- *Failures*(P): the failures of process P.
- *Refusals*(P): $\{(\langle\rangle, r) : Failures(\mathsf{P}) \bullet r\}$ (the *immediate* refusals of P).
- *System*: the behavior process of the configuration.

The use of semantic constraints will be illustrated in chapter 5.

3.7 Validating Descriptions

We have seen 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. For example, 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 to divide 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 described system.

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

---

constraints are not representable directly in CSP: There is no process such that every process that obeys the constraint is a refinement of it. This is because a CSP process must have a fixed alphabet, but the constraints apply to processes which may have different alphabets than each other.

Two criteria for an architectural description that underly all of these analyses 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, no other analysis 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? Which part is to be used in an analysis? An inconsistent description can only lead to trouble.

Because the architectural level of design is fundamentally concerned with 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 structure; if the abstract description is inconsistent, any refinement or implementation of it must necessarily 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 analyzes the communication behavior of a component, but the description used leaves out part of the interface, what good is the analysis? An analysis of contention for a resource can only be applied if *all* of the parts that access that resource are known. How can we decide the timing behavior of a system if the components don't indicate how long they take to compute? We can't.

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. Thus, our discussion of completeness will center on the question of whether a description is complete with respect to a property or analysis of interest.

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 inconsistency or incompleteness, and how simple tests can be applied to guarantee that an architectural description is both consistent and complete. The WRIGHT tests are summarized in figure 3.10. In parentheses, we have indicated the language construct to which each test applies.

3.7.1 Consistency

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.

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. Style Consistency (style)
11. Attachment Completeness (configuration)

Figure 3.10: Summary of tests.

3.7.1.1 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, however, 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 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.

In CSP this informal notion, of one process "obeying the rules" of another process, is captured formally by means of a "refinement relationship" between processes. Formally, refinement is based on the characterization of a processes as the triple (A, F, D) of alphabet, failures and divergences. A process P is *refined by* process Q, written $\mathsf{P} \sqsubseteq \mathsf{Q}$, if their alphabets are the same, the failures of P are a subset of the failures of Q, and the divergences of P are a subset of the divergences of Q. The failures model of CSP, as well as refinement, is discussed in more detail in appendix A.

For the purposes of the WRIGHT checks, the most important property of refinement is that if $\mathsf{P} \sqsubseteq \mathsf{Q}$, then Q must respect all of P's obligations to interact with its environment.

However, where P permits an internal choice among several alternatives, Q may further constrain those choices.

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

1. if $P = (e \rightarrow P \sqcap f \rightarrow P)$ and $Q = (e \rightarrow Q)$ then $P \sqsubseteq Q$
2. if $P = (e \rightarrow P \,\square\, f \rightarrow P)$ and $Q = (e \rightarrow Q)$ then $P \not\sqsubseteq Q$.

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

Returning to the question of port-computation consistency, recall that we do *not* view the collected ports as representing a full specification of the component, which must then be verified. The component will define relations between events at different ports and may also define its behavior in situations that are not expected to arise, due to restrictions defined by the ports. 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.

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 implied by the 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 is a refinement of 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 $\rightarrow$In $\square$ close$\rightarrow$§
  Port Out = $\overline{\text{write!x}}$ $\rightarrow$Out $\sqcap$ $\overline{\text{close}}$$\rightarrow$§
  Computation = (In.read?x $\rightarrow$$\overline{\text{Out.write!(2*x)}}$ $\rightarrow$**Computation**)
    $\square$ (In.close$\rightarrow$$\overline{\text{Out.close}}$$\rightarrow$§)

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 external choice between them in the **Computation** becomes an internal, or non-deterministic choice between the visible events that follow them: ($\overline{\text{Out.write}}$$\rightarrow$**Computation**) $\sqcap$ ($\overline{\text{Out.close}}$$\rightarrow$§). This, of course, matches the port Out.

The second aspect of a port, indicating expectations about the environment, is illustrated by the following example. What if 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$\rightarrow$FailingIn $\square$ close$\rightarrow$§ $\square$ fail$\rightarrow$§

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 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 is not required 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. Such situations occur 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 $\rightarrow$In $\Box$ close$\rightarrow\S$
  Port Out = $\overline{\text{write!x}}$ $\rightarrow$Out $\sqcap$ $\overline{\text{close}}\rightarrow\S$
  Computation = (In.read?x $\rightarrow\overline{\text{Out.write!(2*x)}}$ $\rightarrow$**Computation**)
    $\Box$ (In.close$\rightarrow\overline{\text{Out.close}}\rightarrow\S$)
    $\Box$ (In.fail$\rightarrow\S$)

In this specification, Double promises that if it observes a fail event on port In, then it will terminate immediately, and neither attempt to use In nor 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:

$$\text{CompOut} = (\overline{\text{Out.write!(2*x)}} \rightarrow \text{CompOut}) \sqcap (\overline{\text{Out.close}} \rightarrow \S) \sqcap \S$$

In the projection CompOut, 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. The behavior of the Computation of Double after the event In.fail is not relevant to whether the component is consistent, with either Out or In, because the port In specifies that any interaction on port In must guarantee that fail will not occur. This is an assumption of component Double.

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. Any trace that contains observed events not permitted by the port violates the expectations of the port and need not be considered.

Formally, we can use a *deterministic* version of a process to restrict one process to the traces of another. If we have two processes (with identical alphabets), P and Q, which operate in parallel — P||Q — then the combined process will have no trace that is 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 $det(\mathsf{Q})$ that has exactly the same traces as Q but that is deterministic, then a process 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.

Extracting the observed events is achieved by the CSP project operator (P $\upharpoonright A$). This operator indicates the same process as P, but with all events not in A hidden. The result is an internal (non-deterministic) choice whenever two events, one of them not in A, are possible.

A process is made deterministic via the function $det(\mathsf{P})$. $det(\mathsf{P})$ 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, what event occurs at any point is fully controllable by the environment: $det(\mathsf{P})$ is deterministic. Formally:

Definition 4 *For any process* $\mathsf{P} = (A, F, D)$*,* $det(\mathsf{P}) = (A, F', \varnothing)$ *where* $F' = \{(t, r) \mid t \in Traces(\mathsf{P}) \wedge \forall e : r \bullet t ^\frown \langle e \rangle \notin Traces(\mathsf{P})\}$.

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/Computation 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:*

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

Informally, this means that:

A port specification must be a projection of the **Computation***, under the assumption that all other port interfaces are obeyed by the environment.*

3.7.1.2 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, for example that there are two components in a system, a client and a server. 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. A connector that specifies this "interaction" might be as follows:

Connector Faulty
  Role Client = ($\overline{\text{request}}$→result→Client) ⊓ §
  Role Server = initialize →Operate ...
  Glue = Client.initialize→$\overline{\text{Server.initialize}}$ → **Glue**
    [] Client.request→$\overline{\text{Server.request}}$ → **Glue**
    ...

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. Formally, we define a deadlocking process as one that is not deadlock-free. A deadlock-free process must always either be willing to continue its computation or be in a condition of *successful* termination.[13] Formally:

Definition 5 (Deadlock-Freedom) *A process* $\mathsf{P} = (A, F, D)$ *is deadlock-free if for every trace t such that* $(t, A) \in F$*, last*$(t) = \surd$.

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 initiate event, while the client is only willing to provide a value request: Since neither participant is flexible and they don't agree, deadlock is the 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 represent the behavior of a potential component participant: 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) *If a connector has glue* **G** *and roles* R_1 ... R_n *with processes* P_1 ... P_n*, then the process* $\mathsf{G} \parallel (\parallel i : 1 \,..\, n \bullet \mathsf{R}_i\mathbf{:}\mathsf{P}_i)$ *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** is 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.

[13]Recall that § = $\surd \rightarrow$**STOP**, and so in effect we require that the only stopped process is §.

Initiator Tests

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 initiation is not limited to cases in which one party must define an interaction and the other uses it. If there is a more complicated infrastructure in 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. 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 distinguishes between initiated and observed events precisely to avoid this problem. Recall that in section 3.4, 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: 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. Formally:

Test 4 (Single Initiator) *A connector with glue process* $\mathbf{G}$ *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* $(\alpha\mathbf{G} \cup \alpha\mathsf{R}_1 \cup \ldots \cup \alpha\mathsf{R}_n) - \{\surd\}$. *Further, it must be the case that* $\alpha_i\mathbf{G} \cap \alpha_o\mathbf{G} = \varnothing$ *and* $\forall j : 1..n \bullet \alpha_i\mathsf{R}_j \cap \alpha_o\mathsf{R}_j = \varnothing$.

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. To explain this idea, consider the following extract from a possible WRIGHT specification:

Role Invalid = $\overline{\mathsf{e}} \rightarrow \mathsf{P} \;\Box\; \overline{\mathsf{f}} \rightarrow \mathsf{Q}$....

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: 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 can be detected as a potential deadlock.

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* $t \frown \langle \overline{\mathsf{e}} \rangle \in Traces(\mathsf{P})$, $(t, A \setminus \{\overline{\mathsf{e}}\}) \in F$. [14]

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*.[15]

Concurrent components with a clear substructure that can be analyzed for initiator-commits will arise, for example, when the architecture of a system is described hierarchically, as in section 3.3.3.

3.7.1.3 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.

3.7.1.4 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: Is the name of the instance unique? and, have

[14] Note that this rule applies to components as well as connectors.

[15] This issue will be discussed further in chapter 9.

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. We will have to defer limits on, for example, process parameters to section 3.6 when we can use styles to achieve this effect.

3.7.1.5 Attachments

As we discussed above, an important reason to provide 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 $\mathsf{Out} = \overline{\mathsf{write!x}} \rightarrow \mathsf{Out} \sqcap \overline{\mathsf{close}} \rightarrow \S$

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:

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

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

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

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 to 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

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

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.

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 6 *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 (Compatibility) *A port* P *is compatible with a role* R*, written "*P **compat** R*," if*

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

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.

3.7.1.6 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 3.3.4.3, 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, suppose we have two constraints that contradict each other:

$$\begin{array}{l} \forall\, c : Components;\, p : Ports(c) \bullet Type(p) = DataOutput \\ \wedge\ \exists\, c : Components;\, p : Ports(c) \bullet Type(p) = DataInput \end{array}$$

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 have both types, there can not be any system that obeys both these constraints: The style is inconsistent.

When the constraints are complex, it is easy to generate contradictory constraints. Thus, we highlight the potential for this problem by making an explicit test:

Test 10 (Style Consistency) *There must be at least one configuration that satisfies the style's constraints.*

In this section, we have shown how WRIGHT deals with the issue of consistency in an architectural description by using a series of tests to detect potential inconsistencies. These inconsistencies include misuse of initiated and observed events, deadlocks among interaction participants, interfaces that do not correctly represent the computation of a component, and use of invalid parameters in types, among others.

3.7.2 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.[16]

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.

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 analysis that depends 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. A further discussion of how these additions might be handled appears in chapters 8 and 10.

Another category of completeness 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 can not simply prohibit unattached ports or roles.

To detect this kind of incompleteness while maintaining as much flexibility as possible, we introduce another test on a configuration. If a port or role 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, *in order to be left unattached in a complete configuration*, it must be able to behave as the process that simply halts, *i.e.*, §.

Test 11 (Attachment Completeness) *Every unattached port or role in a configuration must be compatible with, respectively, the role or port* §.

[16]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.

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 so, then users of the style will find their task greatly simplified. We will call such a style a *safe style*. We will not define an explicit test for this property, however, because, while it is desirable, it is not necessary.

Chapter 4

Case Study: AEGIS

4.1 Introduction

In this chapter we show how WRIGHT can be applied to a significant system. This formalization will provide a complete example of the formal specification of the architecture of a software system.

The software architecture describes the structure of the system, decomposing it into independently described components and permitting the analysis of how those components will contribute to achieving the requirements of the overall system. By structuring the system architecturally, the developer hopes to be able to analyze the system at a high level of abstraction and, by increasing independence between components, to provide a means of reducing the complexity of solving each of the parts of the problem. To do so, the architect must specify how the components interact, so that, just as the components are combined to form the overall system, the properties of the components can be combined in analysis to understand the overall system. On a more concrete level, the architecture can also act as a guide to integration of the implementation, where the components represent independently coded entities that are combined by compile- or run-time structures to support the specified interactions.

In a typical software architectural description, the structure of the system is decomposed into a collection of components, each of which is allocated a particular responsibility in the system. This responsibility usually includes a functional component, *i.e.*, what it is required to compute, and some kind of other constraints on its behavior and characteristics. These components, drawn as boxes in an architectural diagram, are combined into a configuration via connectors, drawn as arrows in the diagram. Each of the connectors is given a brief description of what kind of interaction it represents: “dataflow,” “client-server,” “message passing,” *etc.* This description may be used as the basis of some kind of analysis of the system, to justify the design of the system, and to provide a guide to designers of each component and to those responsible for integrating the implemented components.

In our case study, for example, the original architectural description consisted of a diagram of seven components such as “Display Server” and “Doctrine Reasoning,” and arrows representing “client-server” interactions. The components were divided among several research teams, each having the responsibility to implement their components. These modules would then be combined into an executing prototype.

While even an informal software architecture description can provide a means of dividing responsibilities for design and implementation, and may be used for "back-of-the-envelope" analysis of the eventual system, as well as system understanding, the imprecision and lack of semantic basis for the description inherently limits how much it can achieve. When the exact requirements for participation in the interactions are not spelled out, it is likely that integration will require significant alteration of components to meet global assumptions. When the semantics of each component and the interactions is not precise, any analysis of the final system is tentative pending delivery of actual code. The benefits of the high level of abstraction of the architectural description is undermined by the fact that rigor can only be achieve by abandoning the architectural level in favor of less abstract representations.

By formalizing the description of software architecture, WRIGHT permits the software architect to be precise about the meaning of components and connectors while retaining a high level of abstraction. The consistency checks allow the designer to catch problems in the design early, before the components and infrastructure are elaborated in full detail. Further, because the semantics of the description are precisely defined, it is possible to analyze the architecture and ensure that critical properties are obeyed and requirements are met. The activity of careful formalization can also provide opportunities for insight, allowing the designer to recognize opportunities for reuse by exposing commonalities, while highlighting important distinctions that must be made in order for the system to operate successfully. It can also (although we do not discuss it here) provide criteria for acceptance for an implementation of the system.

This case study provides an example of how these benefits can be achieved by carrying out the exercise of formalizing an informal software architecture description and exploring its properties. To do this, we follow in the footsteps of the developers of the AEGIS Prototech demonstration. Our first step is to capture, formally, their initial informal architecture. With this architectural description, now made formal, we will see how analysis in WRIGHT reveals problems that the developers found during the implementation and integration phases. By exposing the problems at an earlier step, formalization reduces the cost of correcting the problems and also means that there are more options in terms of avoiding the problem without unduly complicating the system design.

We will also see how, when formalize the terms that the AEGIS developers used informally, we gain insight into the alternatives that they should have considered explicitly and the trade-offs in selecting between them.

In a second step, we continue to follow the AEGIS developers, considering how we might formalize the architecture that the AEGIS developers used to solve the problems of the initial architecture. We will further show how the formalism elucidates the difference between the first and second systems, and how our formalization of the second system gives confidence that the problems have indeed been resolved. We thus increase the level of confidence in the developer's solution and provide assurance that yet a third redesign phase will not be necessary.

In a third step, we will consider what kind of solutions might have been found if the developers had been able to detect the critical problems at the architectural level. In their final solution, the developers changed a single, simple interaction kind into a collection of four connectors, and a simple pair of component categories into five different special cases (for seven components!). We discuss how the formalization suggests simplifications to this solution that could have provided reuse of infrastructure (both for construction and integration of components) as well as a simpler conceptual model of the system.

Finally, we will revisit the techniques used in the formalization at a more abstract level, showing how we must consider issues such as what information to ignore in order to raise the level of abstraction while retaining critical distinctions (such as between blocking and non-blocking interactions). We will also discuss the idea of how parameterization can be used to increase the generality of descriptions and thus improve the expressive power of the notation.

4.2 The AEGIS "Problem"

The AEGIS Weapons System is a large, complex software system that controls many of the defense functions of modern US Navy ships. As described in one DoD report:

> The AEGIS Weapons Systems (AWS) is an extensive array of sensors and weapons designed to defend a battle group against air, surface and subsurface threats. These weapons are controlled through a large number of control consoles, which provide a wide variety of tactical decision aids to the crew. To manage complexity, the crew can preset conditions under which automated or semi-automated responses occur. This capability is generally referred to as doctrine.

The motivation for using AEGIS as a challenge problem arose through a demonstration exercise of the ARPA Prototyping Technology Program in 1993.[1] Engineers on the real AEGIS system provided a proposed redesign for a part of the system that takes monitored sensor data about moving objects near the ship, and decides what actions to take. To do this the system must resolve the "tracks" of moving objects against its geometrical model of the ship and nearby entities.

An informal description of the proposed architecture of the system is shown in Figure 4.1. The system consists of seven modules. The Experiment Control module provides simulated input from the operator and sensors, as well as a "heartbeat" signal indicating the passage of simulation time. Tracking data is sent to the Track Server, which maintains a record of the currently-monitored moving objects (missiles, other planes, submarines, *etc.*) within its tracking region. The Doctrine Authoring module receives input describing rules of engagement and activation. The GeoServer module takes doctrine information (from the Doctrine Authoring module), and track information (from the Track Server) and, based on its own geometric models, determines which tracks intersect which geometric regions. This information (together with track and doctrine information) is fed to the Doctrine Reasoning module, which determines what action should take place. For the purposes of the prototype these actions, as well as other status information is displayed to the user via a Display Server module. The arrows in the figure indicate the direction of information flow.

[1] The problem was initially presented by Bob Balzer at an ARPA program meeting in Fall 1994. It was later re-presented as a challenge problem at the 1995 Dagstuhl Workshop on Software Architecture [GPT95].

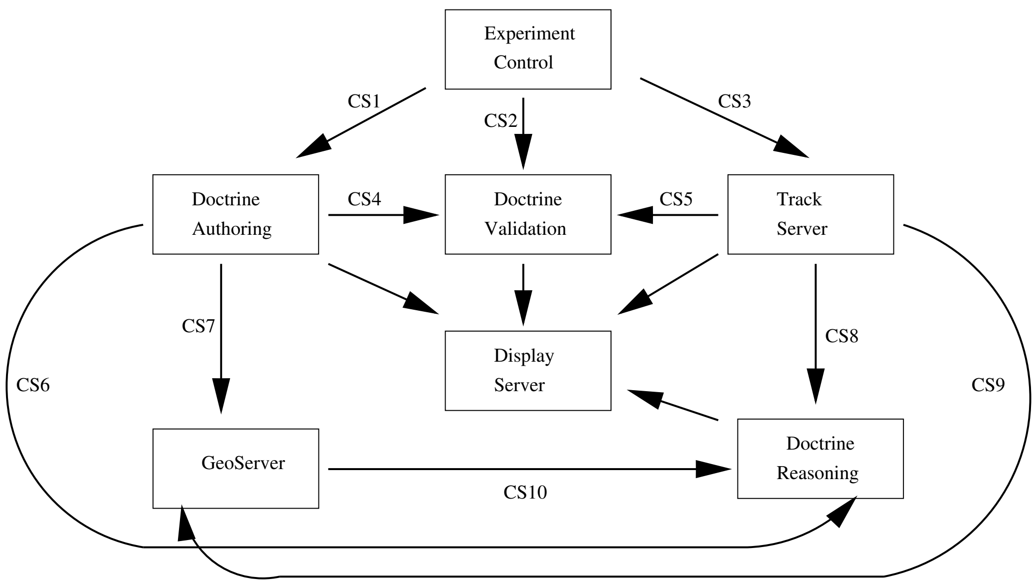

Figure 4.1: The AEGIS prototype architecture.

In the Prototech demonstration, each of the research teams in the program was assigned the task of implementing one or more modules of the system. The modules were to be integrated into a running system that could then be demonstrated for the program sponsors. To make this integration possible the teams had to agree on the nature of the architectural connection that they would use. For implementation reasons (they were building on top of Unix with sockets) they initially agreed to use a uniform client-server organization, in which clients requested data from the servers. Thus information would be "pulled" from the top to the bottom of the figure: *i.e.*, clients at the tip of the arrows, and the servers at the tails. Components that have both incoming and outgoing arrows would act both as a client and a server.

Putting aside internal details of the individual modules, this sounds like a relatively straightforward task. Unfortunately, it turned out to be anything but trivial. First, there were some serious ambiguities about the meaning of client-server interactions. Which party initiated the connection? Was it reestablished after each request? Was the data transferred synchronously? Moreover, the developers discovered, during integration, that there were restrictions induced by implementation choices within modules making it infeasible for certain modules to act both as clients and as servers. Furthermore, the basic design did not account for some advanced monitoring capabilities of the inter-module communication. The net result was that (according to one of the participants) the final integration was something of a nightmare, and the resulting system considerably more complex than had been originally envisioned.

In the remainder of this chapter we use WRIGHT to explore some of these problems. We will focus on a few key problems – primarily those relating to the potential for lost messages. We start by characterizing the naive architectural design. Then we show how this specification must be modified to characterize the "as-built" system.

4.3 The Naive Specification

As noted earlier, the initial model of the AEGIS system proposed by the Prototech developers was to use a client-server model. In this informal client-server model, a client initiates a data request from a server, which fills the requests of each of its clients as they arrive. But this simple, informal description brushes a lot of important information under the rug, and does not provide enough details even to begin a more detailed design. The abstraction doesn't resolve issues such as what protocols are used to make the data request and reply, how termination is signalled, and whether servers must handle multiple requests simultaneously. By expressing this "naive" architectural description in WRIGHT, we will see how these issues come to the fore.

In WRIGHT we begin an architectural description by characterizing the architectural vocabulary with which the system is developed. We will develop each of the elements of the architecture as a type, either of port or role, component, or connector. Each of the type definitions will provide a building block from which each particular system instance can be developed.

4.3.1 Interface Types

The smallest building block in a system is an interface type, used to describe either the interface of a component (port) or the constraints on a participant in an interaction protocol (a connector role).

$$
\begin{array}{l}
\textbf{Interface Type}\ \mathsf{ClientPullT} = \overline{\mathsf{open}} \rightarrow \mathsf{Operate} \sqcap \S \\
\quad \textbf{where}\ \mathsf{Operate} = \overline{\mathsf{request}} \rightarrow \mathsf{result?x} \rightarrow \mathsf{Operate} \\
\qquad\qquad\qquad\qquad \sqcap\ \mathsf{Close} \\
\qquad\qquad \mathsf{Close} = \overline{\mathsf{close}} \rightarrow \S
\end{array}
$$

$$
\begin{array}{l}
\textbf{Interface Type}\ \mathsf{ServerPushT} = \mathsf{open} \rightarrow \mathsf{Operate}\ [\,]\ \S \\
\quad \textbf{where}\ \mathsf{Operate} = \mathsf{request} \rightarrow \overline{\mathsf{result!x}} \rightarrow \mathsf{Operate} \\
\qquad\qquad\qquad\qquad [\,]\ \mathsf{Close} \\
\qquad\qquad \mathsf{Close} = \mathsf{close} \rightarrow \S
\end{array}
$$

The ClientPullT is the basic type used for the ports of a client component (and for the role in a connector which will be played by such a component). The ClientPullT process indicates that a client will begin by establishing the connection with the open event. (Recall that the overbar notation ($\overline{\mathsf{open}}$) indicates that the client initiates the event.) Note how the formal specification already provides important information about the system's behavior. ClientPullT includes in its initial state an internal choice (indicated by $\sqcap$) of the process $\S$. Thus, the interface type specifies that a client is *not required* to open a connection if it does not plan to use a given server.

After opening the connection, an operational phase is begun, in which the client can repeatedly choose to request data. The client expects to receive exactly one result for each request. At any time the client may choose to close the connection, after which the interaction ceases (as indicated by the $\S$ process). Again, the use of internal choice between request and close indicates that it is the *client* that controls how many times a request is made.

The ServerPushT process is the complement of the ClientPullT. The server expects another party to open the connection (it *observes* this event, as indicated by the absence of an overbar). Then, it will repeatedly provide responses to requests until it recognizes a close event, after which it is free to terminate. Use of external choice ($\Box$) indicates that a server is required to handle either alternative. Thus, while the client had the option of either requesting or closing, the server is required to handle both requests and closes.

Consistency

As we discussed in chapter 3, WRIGHT defines checks that can locate inconsistencies in a specification. Thus, for each of the definitions that we use in this case study, we will indicate what checks apply at each stage of the specification. Although none of the definitions in this chapter is, in fact, inconsistent, consistency checking is an important step in any WRIGHT specification effort. We used the checks to locate problems in earlier drafts of our AEGIS specification, and in chapter 6 we will see how they provide a powerful analytic tool.

To be used as a role, each interface type must be shown to be deadlock-free (test 3). Both ClientPullT and ServerPushT are deadlock-free. Both also obey "initiator-commits" (test 5).

4.3.2 Connectors

In combination (or even singly), the ClientPullT and ServerPushT interface processes would seem to be adequate to define the client-server interaction; each initiated event in one process corresponds to an observed event in the other. The ClientServer connector specification confirms this relation between events; each line of the **Glue** specification indicates the correspondence between a pair of events – when the client initiates an open, the server will subsequently observe an open, and so on.[2] This is a very common case in architectural connectors, and many descriptive notations specialize their connector descriptions to it (*e.g.*, [LAK+95, YS97]). As we will see later, however, there are other cases where there may be more complex relationships, involving partial visibility of events or run-time mechanisms (that are not part of the abstract computation) that require a more complex **Glue**. WRIGHT requires that even the "trivial" glue be spelled out in full, although it could easily be generated automatically. (See chapter 9 for a discussion of this issue.)

Connector ClientServer =
  Role Client = ClientPullT
  Role Server = ServerPushT
  Glue = Client.open $\rightarrow \overline{\text{Server.open}} \rightarrow$ **Glue**
    $\Box$ Client.close $\rightarrow \overline{\text{Server.close}} \rightarrow$ **Glue**
    $\Box$ Client.request $\rightarrow \overline{\text{Server.request}} \rightarrow$ **Glue**
    $\Box$ Server.result?x $\rightarrow \overline{\text{Client.result!x}} \rightarrow$ **Glue**
    $\Box$ §

[2] Remember, the roles specify *component* behavior, while the **Glue** specifies the *connector* behavior. Thus, initiated events in roles appear as observed events in the **Glue**, and vice versa.

Because ClientServer is such a simple connector, it is easy to check that it obeys each of the consistency constraints on a connector: deadlock-freedom (test 2), single-initiator (test 4), and initiator-commits (test 5).

However, this connector also illustrates the importance of the consistency checks. Suppose that we had specified the ClientPullT and ServerPushT interface types separately and made different choices about them. For example, an alternate client interface might have omitted the requirement that a client signal the close of the connection:

$$\begin{aligned}
&\textbf{Interface Type}\ \mathsf{AltClientPullT} = \overline{\mathsf{open}} \rightarrow \mathsf{Operate} \sqcap \S \\
&\quad \textbf{where}\ \mathsf{Operate} = \overline{\mathsf{request}} \rightarrow \mathsf{result?x} \rightarrow \mathsf{Operate} \\
&\qquad\qquad\qquad\qquad \sqcap\ \mathsf{Close} \\
&\qquad\qquad \mathsf{Close} = (\overline{\mathsf{close}} \rightarrow \S) \sqcap \S
\end{aligned}$$

This is a seemingly small change, and might be reasonable: If the client is about to terminate anyway, how much trouble can be caused by not closing? But replacing ClientPullT with AltClientPullT in the ClientServer connector reveals a problem: If the client does not close the connection, the server has no information about when it can terminate, and thus must remain in existence indefinitely. The ServerPushT role indicates that the server can not terminate until after a close event, but this will never be supplied by the AltClientPullT. This is detected by WRIGHT's consistency check for deadlock (test 2). In particular, the check would indicate that after the event sequence $\langle$Client.open, Server.open$\rangle$, the Client may accept only the $\surd$ event while the Server expects to be supplied with Server.close or Server.request. The **Glue** can not supply either of those events until the Client permits it, and so a deadlock may occur.

Our specification of the ClientServer connector type also answers several important questions left unanswered by the developer's informal description of the AEGIS architecture. It indicates that clients and servers interact by passing data from clients to servers, but that clients control when connections are initialized (by the open event) and how many times requests are made. It further specifies key constraints on clients. For example, if a client is not going to make any more requests, it must close the connection so that a server may terminate. The connector also indicates important properties beyond the basic data-passing functionality. For example, requests and results are paired one-to-one: message delivery in AEGIS is assumed to be *reliable*, with no dropped messages.

4.3.3 Components

In a client-server system such as AEGIS, there are three kinds of components: those that act as clients (*e.g.*, DoctrineValidation), those that act as servers (*e.g.*, ExperimentControl), and those that combine the two functions (*e.g.*, GeoServer). Components of each of these kinds can have different numbers of ports, and so we represent them in WRIGHT using parameterized types.

Consider the Client component type, shown in figure 4.2. The Client component type is straightforward. It has complete control over its actions at any time, as long as it obeys the ClientPullT protocol on each of its ports. For simplicity, we assume that it begins by opening each of its connections, and finishes by closing each of them. During the middle

$$
\begin{array}{l}
\textbf{Component}\ \mathsf{Client}(\mathit{numServers} : 1..) = \\
\quad \textbf{Port}\ \mathsf{Service}_{1..\mathit{numServers}} = \mathsf{ClientPullT} \\
\quad \textbf{Computation} = (;\, x : 1..\mathit{numServers} \bullet \overline{\mathsf{Service}_x.\mathsf{open}} \rightarrow \S)\ ;\ \mathsf{UseOrExit} \\
\quad \textbf{where}\ \mathsf{UseOrExit} = \mathsf{UseService} \sqcap \mathsf{Exit} \\
\qquad\quad \mathsf{UseService} = \sqcap\, x : 1..\mathit{numServers} \\
\qquad\qquad\qquad\qquad \bullet\ \overline{\mathsf{Service}_x.\mathsf{request}} \rightarrow \mathsf{Service}_x.\mathsf{result?y} \rightarrow \mathsf{UseOrExit} \\
\qquad\quad \mathsf{Exit} = (;\, x : 1..\mathit{numServers} \bullet \overline{\mathsf{Service}_x.\mathsf{close}} \rightarrow \S)\ ;\ \S
\end{array}
$$

Figure 4.2: **Component** Client.

$$
\begin{array}{l}
\textbf{Component}\ \mathsf{Server}(\mathit{numClients} : 1..) = \\
\quad \textbf{Port}\ \mathsf{Client}_{1..\mathit{numClients}} = \mathsf{ServerPushT} \\
\quad \textbf{Computation} = \mathsf{WaitForClient}_{\{\},\{\}} \\
\qquad \textbf{where}\ \mathsf{WaitForClient}_{O,C} = \square\, x : ((1..\mathit{numClients}) \setminus (O \cup C)) \\
\qquad\qquad\qquad\qquad\qquad \bullet\ \mathsf{Client}_x.\mathsf{open} \rightarrow \mathsf{DecideNextAction}_{O \cup \{x\},C}
\end{array}
$$

$$
\mathsf{DecideNextAction}_{O,C} = \begin{cases}
\mathsf{WaitForClient}_{O,C} \sqcap (\sqcap\, x : O \bullet \mathsf{ReadFromClient}_{x,O,C}), \\
\qquad \textbf{when}\ O \neq \{\} \wedge O \cup C \neq (1..\mathit{numClients}) \\[1ex]
\sqcap\, x : O \bullet \mathsf{ReadFromClient}_{x,O,C}, \\
\qquad \textbf{when}\ O \neq \{\} \wedge O \cup C = (1..\mathit{numClients}) \\[1ex]
\mathsf{WaitForClient}_{\{\},C}, \\
\qquad \textbf{when}\ O = \{\} \wedge C \neq (1..\mathit{numClients}) \\[1ex]
\S, \qquad \textbf{when}\ O = \{\} \wedge C = (1..\mathit{numClients})
\end{cases}
$$

$$
\begin{array}{rl}
\mathsf{ReadFromClient}_{x,O,C} = & \mathsf{Client}_x.\mathsf{request} \rightarrow \overline{\mathsf{Client}_x.\mathsf{result!y}} \rightarrow \mathsf{DecideNextAction}_{O,C} \\
& \square\ \mathsf{Client}_x.\mathsf{close} \rightarrow \mathsf{DecideNextAction}_{O \setminus \{x\}, C \cup \{x\}}
\end{array}
$$

Figure 4.3: **Component** Server.

phase, the process UseService selects from among its connections to request a new data item. The choice is entirely up to the client, as indicated by the use of internal choice.

Now consider the specification of the Server component type (shown in figure 4.3). A Server component provides data services to one or more clients. With each client, the server uses the ServerPushT protocol. At any point in the protocol, each client is in one of three states: "Open" (represented by the set O), "Closed" (represented by the set C), or "not yet Open" (all others).

The **Computation** specification illustrates a number of critical issues about the AEGIS architecture. What mechanisms are available for the server to locate new connections that should be opened? To receive a client's request? When can a newly closed client connection be recognized, and what action should be taken? The Server specification provides answers to these questions.

The **Computation**, shown in figure 4.3, describes a server that can handle at most one client request at a time. This is indicated by the internal choice, in the definition of

DecideNextAction, among versions of the process ReadFromClient. The server may also choose to wait for an open request from any of a set of clients. This interaction pattern, of selecting a single client for a request or a set of clients for an open, is characteristic of the Unix socket mechanism, which the developers selected as an implementation base. Our formalization permits us to see the consequences of this choice, while abstracting other implementation details, in its effects on the server component type.

Because of the blocking open and request protocol, as well as the requirement that the server eventually handle all requests, the server must keep track of the statuses of the different clients. Some have not yet opened, and the server can wait for them to open; some have opened but not closed, and the server can expect either a request or a close from them; and some have closed, and the server must not expect any further action from them. In our WRIGHT specification, the open and the closed clients are represented by the state variables O and C respectively (those that have never opened are members of the set $(1..\textit{numClients}) \setminus (O \cup C)$). Given these different statuses, there are four distinct cases for the server, requiring different choices of action: The least constrained case is when there are both open and unopened clients; in this case, the server is free to make any choice of action. If every client has already opened, the server must not wait for a client to open. If no client is open, then the server does not have the option of waiting for a request. Once every client has closed, the only possible action by the server is to terminate. Each of these different statuses is represented concisely in WRIGHT using a conditional process definition. There is a different definition of DecideNextAction for each case, guarded by a predicate over its state. For example, the case where there are no clients that have not yet opened and at least one client still open is indicated by the predicate $O \neq \{\} \wedge O \cup C = (1..numClients)$.

A number of components in the AEGIS architecture combine the properties of a Client and a Server. We formalize this case in the component type MixedComp (figure 4.4). It must deal with open, close, and request events from its clients, but it also has the option of requesting a service from one of its servers at any time.

Note that, when specifying a component formally, we define *all* of its interactions, not just those that are considered "primary." If a component acts as a server, we specify not only what its protocol is with its clients (which is important), but also whether it depends on the existence of other components to carry out its computation (which is equally important to the success of the system). In AEGIS, we have shown how a "server" component in the informal specification (*e.g.*, the TrackServer) may actually be a "client" of another part of the system. In this way, a formal specification reveals properties of the architecture that an informal specification may leave implicit.

Consistency

As for connectors, we can apply WRIGHT's consistency checks to the component types. There is one consistency check that applies to a parameterized component type: we must show that port-computation consistency (test 1) holds for any legal integer parameters. Also, we can observe a property of using **Interface Types** to specify both components and connectors: They greatly simplify attachment checks in configurations. In particular, at this point we know that any use of a Client's or a MixedComp's Service port as the Client role in a ClientServer connector will be compatible (test 8). Similarly, any use of a Server's

Component MixedComp($numServers : 1..;\ numClients : 1..$) =
  Port $\mathsf{Service}_{1..numServers}$ = ClientPullT
  Port $\mathsf{Client}_{1..numClients}$ = ServerPushT
  Computation = OpenServices ; $\mathsf{WaitForClient}_{\{\},\{\}}$
    where

$$\mathsf{WaitForClient}_{O,C} = \square\, x : ((1..numClients) \setminus (O \cup C)) \bullet \mathsf{Client}_x.\mathsf{open} \rightarrow \mathsf{DecideNextAction}_{O \cup \{x\},C}$$

$$\mathsf{DecideNextAction}_{O,C} = \begin{cases} \mathsf{WaitForClient}_{O,C} \sqcap (\sqcap\, x : O \bullet \mathsf{ReadFromClient}_{x,O,C}) \\ \quad \sqcap (\mathsf{UseService};\ \mathsf{DecideNextAction}_{O,C}), & \textbf{when } O \neq \{\} \wedge O \cup C \neq (1..numClients) \\ (\sqcap\, x : O \bullet \mathsf{ReadFromClient}_{x,O,C}) \sqcap (\mathsf{UseService};\ \mathsf{DecideNextAction}_{O,C}), & \textbf{when } O \neq \{\} \wedge O \cup C = (1..numClients) \\ \mathsf{WaitForClient}_{\{\},C} \sqcap (\mathsf{UseService};\ \mathsf{DecideNextAction}_{\{\},C}), & \textbf{when } O = \{\} \wedge C \neq (1..numClients) \\ (\mathsf{UseService};\ \mathsf{DecideNextAction}_{\{\},(1..numClients)}) \sqcap \mathsf{Exit}, & \textbf{when } O = \{\} \wedge C = (1..numClients) \end{cases}$$

$$\begin{aligned} \mathsf{ReadFromClient}_{x,O,C} = {} & \mathsf{Client}_x.\mathsf{request} \rightarrow (\mathrm{OptionalUseService}\ ; \\ & \qquad \overline{\mathsf{Client}_x.\mathsf{result!y}} \rightarrow \mathsf{DecideNextAction}_{O,C}) \\ & \square\ \mathsf{Client}_x.\mathsf{close} \rightarrow \mathsf{DecideNextAction}_{O \setminus \{x\}, C \cup \{x\}} \end{aligned}$$

$$\mathsf{UseService} = \sqcap\, x : (1..numServers) \bullet \overline{\mathsf{Service}_x.\mathsf{request}} \rightarrow \mathsf{Service}_x.\mathsf{result?y} \rightarrow \S$$

$$\mathsf{OptionalUseService} = (\mathrm{UseService}\ ;\ \mathrm{OptionalUseService}) \sqcap \S$$

$$\mathsf{OpenServices} = ;\ x : (1..numServers) \bullet \overline{\mathsf{Service}_x.\mathsf{open}} \rightarrow \S$$

$$\mathsf{Exit} = ;\ x : (1..numServers) \bullet \overline{\mathsf{Service}_x.\mathsf{close}} \rightarrow \S$$

Figure 4.4: **Component** MixedComp.

or MixedComp's Client port in a Server role will be compatible.

4.3.4 Configuration

Now that we have described the basic vocabulary of the naive AEGIS architecture, we can describe the configuration of the testbed system.[3] It is shown in figure 4.5. The instantiation and attachment declarations are a direct textual transcription of the informal architectural diagram in figure 4.1, and that diagram can be still be used. However, as we will see in the next section, the additional information in the WRIGHT specification (namely the component and connector type specifications) will permit us to analyze the architecture initially proposed by the AEGIS developers and discover problems that they found only late in the process of implementation.

Consistency and Completeness

For the instances, there is one consistency check required: the parameter range check (test 7). Each of the component types requires that its parameters be positive integers, and we have supplied these. As we noted earlier, the attachments are all between identically defined ports and roles, and so port-role compatibility is satisfied (test 8). There are no dangling ports or roles in the configuration, and so the configuration is complete (test 11).

4.4 Analyzing and Changing the Specification

The WRIGHT specification described in the previous section is a reasonable and useful description of the architecture of the AEGIS system as it was initially envisioned by the developers. The protocol described in the ClientServer connector and the computation patterns covered by the connectors Client, Server, and MixedComp describe the high level design of the system, defining the computation model and the requirements on the run-time infrastructure for the proposed system. By making precise the abstractions and intuitions of the informal specification, our WRIGHT specification clarifies the meaning of the architecture and ensures consistency between different parts of the specification.

As we will now see, we can use the specification to discover a number of shortcomings of the initial design, shortcomings that led the developers to a major reworking of the system and that seriously complicated the final product. The system as it was eventually constructed bore little resemblance to the simple client-server system described above. In this section, we look at some of the issues that arose in the AEGIS design, show how they are exposed by the preceding WRIGHT description, and further show how the solutions found by the AEGIS team can be expressed in WRIGHT, thus ensuring that the architectural description matches the system as built.

[3] For reasons that will become clearer later this initial description excludes the DisplayServer. In the next section we include it in the specification.

Configuration Testbed
[..Type definitions..]
Instances
ExperimentControl : Server(3)
DoctrineAuthoring : MixedComp(1,3)
DoctrineValidation : Client(3)
TrackServer : MixedComp(1,3)
GeoServer : MixedComp(2,1)
DoctrineReasoning : Client(3)
$CS_{1..10}$: ClientServer
Attachments
ExperimentControl.Client **as** CS_1.Server
DoctrineAuthoring.Service **as** CS_1.Client
ExperimentControl.Client **as** CS_2.Server
DoctrineValidation.Service **as** CS_2.Client
ExperimentControl.Client **as** CS_3.Server
TrackServer.Service **as** CS_3.Client
DoctrineAuthoring.Client **as** CS_4.Server
DoctrineValidation.Service **as** CS_4.Client
TrackServer.Client **as** CS_5.Server
DoctrineValidation.Service **as** CS_5.Client
DoctrineAuthoring.Client **as** CS_6.Server
DoctrineReasoning.Service **as** CS_6.Client
DoctrineAuthoring.Client **as** CS_7.Server
GeoServer.Service **as** CS_7.Client
TrackServer.Client **as** CS_8.Server
DoctrineReasoning.Service **as** CS_9.Client
TrackServer.Client **as** CS_9.Server
GeoServer.Service **as** CS_9.Client
GeoServer.Client **as** CS_{10}.Server
DoctrineReasoning.Service **as** CS_{10}.Client
End Testbed.

Figure 4.5: The testbed configuration.

$$
\begin{array}{l}
\textbf{Interface Type}\ \mathsf{ClientPushT} = \overline{\mathsf{open}} \rightarrow \mathsf{Operate} \sqcap \S \\
\quad \textbf{where}\ \mathsf{Operate} = \overline{\mathsf{request!x}} \rightarrow \mathsf{result} \rightarrow \mathsf{Operate} \\
\qquad\qquad\qquad\quad \sqcap\ \mathsf{Close} \\
\qquad\quad \mathsf{Close} = \overline{\mathsf{close}} \rightarrow \S
\end{array}
$$

$$
\begin{array}{l}
\textbf{Interface Type}\ \mathsf{ServerPullT} = \mathsf{open} \rightarrow \mathsf{Operate}\ [\,]\ \S \\
\quad \textbf{where}\ \mathsf{Operate} = \mathsf{request?x} \rightarrow \overline{\mathsf{result}} \rightarrow \mathsf{Operate} \\
\qquad\qquad\qquad\quad [\,]\ \mathsf{Close} \\
\qquad\quad \mathsf{Close} = \mathsf{close} \rightarrow \S
\end{array}
$$

$$
\begin{array}{l}
\textbf{Connector}\ \mathsf{ClientServerPush} = \\
\quad \textbf{Role}\ \mathsf{Client} = \mathsf{ClientPushT} \\
\quad \textbf{Role}\ \mathsf{Server} = \mathsf{ServerPullT} \\
\quad \textbf{Glue} = \mathsf{Client.open} \rightarrow \overline{\mathsf{Server.open}} \rightarrow \textbf{Glue} \\
\qquad\qquad [\,]\ \mathsf{Client.close} \rightarrow \overline{\mathsf{Server.close}} \rightarrow \textbf{Glue} \\
\qquad\qquad [\,]\ \mathsf{Client.request?x} \rightarrow \overline{\mathsf{Server.request!x}} \rightarrow \textbf{Glue} \\
\qquad\qquad [\,]\ \mathsf{Server.result} \rightarrow \overline{\mathsf{Client.result}} \rightarrow \textbf{Glue} \\
\qquad\qquad [\,]\ \S
\end{array}
$$

Figure 4.6: The ClientServerPush connector.

4.4.1 Issue: Direction of Data Flow

One of the issues that is highlighted by the formal description of the AEGIS system is directionality of dataflow. The AEGIS system contains a component, the "Display Server," that the initial informal architecture described as a server. However, this component does not match the Server component type because it does not supply data, but instead receives it. It is also not a Client, because it does not initiate the interchange of data but instead waits until data is available. By formalizing one intuitive notion of client-server interaction in the ClientServer connector type, we reveal an ambiguity: Usually, data flows from server to client, but sometimes it flows in the opposite direction, from client to server. The informal specification does not make this clear.

To describe the less common situation, of data flow from client to server, we specify a second connector, ClientServerPush, together with corresponding new port/role declarations (see figure 4.6). (The term Push is used to indicate that the client *pushes* data toward the server, rather than pulling it from the server.) This new connector clarifies how DisplayServer interacts with the rest of the system.

4.4.2 Issue: Potential for Deadlock in Servers

A more serious issue, that created significant problems for the AEGIS developers, is the issue of how servers handle requests from multiple clients. During the integration of their components, they discovered a subtle flaw: While they were using a supposedly reliable message passing protocol, servers were not responding to client's requests. It appeared that messages were being lost.

In fact, the problem is a result of their selection of an implementation platform. During their architectural design, the developers selected a Unix socket mechanism to carry requests and results between clients and servers. By relying on an informal characterization of the client-server interaction and the effect of the Unix socket mechanism on their components, they failed to recognize the interact between two parts of the architecture that lead to a problem: Unless it is internally concurrent, a server can only wait for one socket request at a time. In effect, the server must *guess* which of the clients will be the next one to make a request.

The informal specification of the architecture provided no information about how a server can use the client-server connector, and so the developers were unable to locate this problem until testing, where they discovered that clients were not being served. In our WRIGHT specification, by formalizing the nature of the interaction between clients and servers, and by specifying the effects of the implementation platform choice on the servers' computations, we discover this problem at the architectural phase, before the choice of platform has been committed and implementation effort has begun.

In our specification, the nature of a server's interaction with its clients is described in its **Computation** and the client-server connector type. The fact that a server may not satisfy a client's request, despite the requirement that it do so, is detected as a potential deadlock in a system that uses the Server component as described above. The WRIGHT specification shows the property of blocking requests to the socket mechanism by the use of an internal choice over the set of request events in the Server (and MixedComp) specification. This indicates that the servers are free to handle any of their clients, excluding the other clients while doing so.

A server does not, however, deadlock on its own: If the clients are able to fulfill their obligation either to request or close, then no problems occur. Deadlock can be a problem, however, when more than one client and server are involved. Consider a simplified system topology with two servers, S1 and S2, and two clients C1 and C2, in which both clients interact with both servers. (One such pattern occurs in the system with components DoctrineAuthoring, TrackServer, GeoServer, and DoctrineReasoning.) What happens if client C1 plans to make a request first to S1 and then S2, while client C2 makes a request first to S2 and then S1? If S1 and S2 both guess wrong about which component will make the first request (*i.e.*, S1 guesses C2 and S2 guesses C1), then the system will deadlock. Neither service can proceed before the other, since each is waiting for the other client, which is itself waiting for the other server.

The informal specification provided no information about how a server handles multiple clients; it is only by formalizing the architecture that we can discover the problem at this stage. By providing a precise characterization of the architecture, we can detect what turned out to be a difficult problem to fix during implementation at a stage where it is still easy to repair.

4.4.2.1 Using Dynamic Connections

The AEGIS designers took three approaches to solving the server deadlock problem. The first takes advantage of the fact that it is possible to wait for a *set* of clients to open connections. The protocols are changed so that an open event precedes *every* client

Interface Type DClientPullT = $\overline{\mathsf{open}} \rightarrow \overline{\mathsf{request}}$
$\rightarrow \mathsf{result?x} \rightarrow \overline{\mathsf{close}}$
$\rightarrow \mathsf{DClientPullT}$
$\sqcap$ §
Interface Type DServerPushT = $\mathsf{open} \rightarrow \mathsf{request}$
$\rightarrow \overline{\mathsf{result!x}} \rightarrow \mathsf{close}$
$\rightarrow \mathsf{DServerPushT}$
$\square$ §

Figure 4.7: Interface types for dynamic interaction.

Component DynamicServer (*numClients* : 1..) =
Port $\mathsf{Client}_{1..numClients}$ = DServerPushT
Computation = WaitForClient $\square$ §
where WaitForClient = $\forall\, i : 1..numClients$
$\square\ \mathsf{Client}_i.\mathsf{open}$
$\rightarrow \mathsf{Client}_i.\mathsf{request}$
$\rightarrow \overline{\mathsf{Client}_i.\mathsf{result!x}}$
$\rightarrow \mathsf{Client}_i.\mathsf{close} \rightarrow$ **Computation**

Figure 4.8: The DynamicServer component.

request (see figure 4.7).[4] These protocols are used to make a DynamicServer that waits for open events rather than request events, as in figure 4.8. What this does, in effect, make initialization of the connection implicit in the first open, and change a client's request for data contain three events, open then request then close. (The result still consists of a single action.) There can still be ongoing state kept by a server about the client, because the server knows that a close event does *not*, anymore, mean that the client is never going to request again.

This problem can also arise with the service request portion of a MixedComp, and this is solved by *serverizing* a mixed computation. That is, instead of using a ClientPullT to wait for data, the component uses a ServerPullT port to receive notification when data is available. A serverized component is shown in figure 4.9.

By precisely specifying alternative interaction patterns, we can consider their effect on a proposed configuration. If we change a server to use dynamic connections, then the WRIGHT checks show us that we must also change any clients, because the role of the default ClientServer and the new DClientServer connector has changed from ClientPullT to DClientPullT. By using precise definitions and applying the standard consistency checks, we ensure that the effect of a change is predictable and does not result in new inconsistencies of the system.

Further, we can analyze the new DynamicServer and DynamicServerized components to show that they do not have the problem that prompted the change: servers do not need

[4] Similar definitions for DClientPushT and DServerPullT are not shown. Also, connectors DClientServer and DClientServerPush are straightforward but omitted for brevity.

Component DynamicServerized (*numServers* : 0..; *numClients* : 0..) =
  Port $\text{Service}_{1..numServers}$ = DServerPullT
  Port $\text{Client}_{1..numClients}$ = DServerPushT
  Computation = WaitForService [] WaitForClient [] §
  where WaitForService =
    $\forall\, i : 1..numServers$
      [] $\text{Service}_i\text{.open} \rightarrow \text{Service}_i\text{.request?x}$
      $\rightarrow \overline{\text{Service}_i\text{.result}} \rightarrow \text{Service}_i\text{.close}$
      $\rightarrow$ **Computation**
    WaitForClient =
    $\forall\, i : 1..numClients$ [] $\text{Client}_i\text{.open}$
      $\rightarrow \text{Client}_i\text{.request}$
      $\rightarrow \overline{\text{Client}_i\text{.result!x}}$
      $\rightarrow \text{Client}_i\text{.close}$
      $\rightarrow$ **Computation**

Figure 4.9: The DynamicServerized component.

Connector OpenLoopBuffer =
  Role Source = ClientPushT
  Role Target = ClientPullT
  Glue = OpenPhase ; $\text{Operate}_{\langle\rangle}$
    where OpenPhase = Source.open $\rightarrow \overline{\text{Target.open}} \rightarrow$ § [] Target.open $\rightarrow$ Source.open $\rightarrow$ §
      $\text{Operate}_{\langle\rangle}$ = Source.request?x $\rightarrow \overline{\text{Source.result}} \rightarrow \text{Operate}_{\langle x\rangle}$
        [] Target.request $\rightarrow$ WaitForData
      $\text{Operate}_{S+\langle x\rangle}$ = Source.request?y $\rightarrow \overline{\text{Source.result}} \rightarrow \text{Operate}_{\langle y\rangle+S+\langle x\rangle}$
        [] Target.request $\rightarrow \overline{\text{Target.result!x}} \rightarrow \text{Operate}_S$
      WaitForData = Source.request?x $\rightarrow \overline{\text{Target.result!x}} \rightarrow \overline{\text{Source.result}} \rightarrow \text{Operate}_{\langle\rangle}$

Figure 4.10: An Open-loop buffer.

to guess what client will make a request, and thus deadlock is not a problem. The WRIGHT specification helps us be confident that this is indeed a viable solution.

4.4.2.2 Avoiding Unnecessary Synchronizations

A second approach to avoiding a server blocking on a request is to permit the server to provide the data asynchronously, in anticipation of a request. This is achieved through the use of an OpenLoopBuffer connection (figure 4.10), which guarantees that the source of data will never block waiting for the target to become ready. The buffer stores data until the target requests it, or blocks the target until the source makes new data available. (Such a connection is essentially a pipe.)

Notice how the WRIGHT **Glue** mechanism permits this interaction to be described *without modifying the component interfaces*. Unlike the DClientServer connector, this connector can replace a ClientServer connector without modifying the data target, or a ClientServerPush connector without modifying the data source.

Component ThreadedMixedComp (*numPushServers* :0..; *numPullServers* : 0 .. ;
numDPushServers :0..; *numDPullServers* :0..;
numPullClients : 0..; *numPushClients* : 0..;
numDPullClients : 0..; *numDPushClients* : 0..;
) =
Port $\mathsf{PushServer}_{1..numPushServers}$ = ServerPushT
Port $\mathsf{PullServer}_{1..numPullServers}$ = ServerPullT
Port $\mathsf{DPushServer}_{1..numDPushServers}$ = DServerPushT
Port $\mathsf{DPullServer}_{1..numDPullServers}$ = DServerPullT
Port $\mathsf{PushClient}_{1..numPushClients}$ = ClientPushT
Port $\mathsf{PullClient}_{1..numPullClients}$ = ClientPullT
Port $\mathsf{DPushClient}_{1..numDPushClients}$ = DClientPushT
Port $\mathsf{DPullClient}_{1..numDPullClients}$ = DClientPullT
Computation = $\forall\, i : 1..numPushServers \parallel \mathsf{PushServer}_i$**:**ServerPushT
$\parallel \forall\, i : 1..numPullServers \parallel \mathsf{PullServer}_i$**:**ServerPullT
$\parallel \forall\, i : 1..numDPushServers \parallel \mathsf{DPushServer}_i$**:**DServerPushT
$\parallel \forall\, i : 1..numDPullServers \parallel \mathsf{DPullServer}_i$**:**DServerPullT
$\parallel \forall\, i : 1..numPushClients \parallel \mathsf{PushClient}_i$**:**ClientPushT
$\parallel \forall\, i : 1..numPullClients \parallel \mathsf{PullClient}_i$**:**ClientPullT
$\parallel \forall\, i : 1..numDPushClients \parallel \mathsf{DPushClient}_i$**:**DClientPushT
$\parallel \forall\, i : 1..numDPullClients \parallel \mathsf{DPullClient}_i$**:**DClientPullT

Figure 4.11: A multi-threaded solution.

4.4.2.3 Multi-threading Components

The third and final approach used by the Prototech team to avoid server deadlocks is perhaps the most flexible: to alter the server's implementation mechanism so that it can handle multiple connections at once. If we use a multi-threaded implementation (easily represented in CSP using the || operator), a single component type can use all of the possible interface protocols (see figure 4.11).[5] We represent this in WRIGHT by parameterizing a component by the number of each type of port. Then, for each port the **Computation** has a parallel thread to carry out the interaction at that port.

This solution is used for two of the components, TrackServer and ExperimentControl. The instance declarations for these are as follows:

TrackServer : ThreadedMixedComp (3,0,1,0,1,0,0,0)
ExperimentControl : ThreadedMixedComp (1,0,0,0,0,0,0,2)

TrackServer has three static push server ports, one dynamic push server port, and one static pull client port. Thus, its **Computation** will have five sub-processes executing in parallel, one for each port. ExperimentControl has one static push server port and two dynamic push client ports, implemented by three independent processes.

By precisely specifying the three solutions to the server deadlock problem, we can see how each of the solutions has benefits and drawbacks. The "dynamic connection"

[5]While CSP makes this look like the simplest solution of all, depending on the implementation base it may require a complex implementation.

solution has the advantage of allowing system developers to continue to use the original infrastructure platform, but this solution means that a change to one component requires a change in each of the components with which it interacts. Using an OpenLoopBuffer requires changes to only one of the component's interfaces, but moves away from the client-server model. In addition, it requires the development of infrastructure to support buffering and may have unfortunate consequences if synchronization of components is a concern (data may be arbitrarily delayed between components). The approach of multi-threading is the most flexible, retaining the simple client-server model and not requiring the "serverization" of components. But, it may be difficult to implement since it requires the use of concurrency within the component.

Capturing these different solutions at the architectural level provides the opportunity to select the best solution for the entire system. If the infrastructure is available, multi-threading can be used throughout to retain a close correspondence with the original solution. If not, but delays in data delivery can be tolerated, buffering can be used (thus in effect converting the system to a pipe-filter style). There are also situations where dynamic connections and serverized components are the right choice.

The actual Prototech solution is shown in figures 4.12 and 4.13. Note that it incorporates all of the three solutions in different places, rather than only one. Because the Prototech team only discovered these problems at the *implementation* phase of their development lifecycle, after responsibility for different components had already been divided and implementation platform decisions bound, they were not able to realize that a common solution might have been better than the hybrid approach that they took. Because the specification of the architecture was informal, each team chose to interpret it in a different way. This resulted in inconsistencies of approach and significant roadblocks to the system's eventual integration.

Their non-uniform approach, resulting from the failure to precisely specify the architecture and analyze it for problems that they would later encounter, also meant that it was difficult to handle concerns such as instrumentation of the testbed in a uniform way.

4.4.3 Issue: Instrumenting Communication

An important issue for the AEGIS testbed is that it is an experimental system. As such, there is a requirement that the interactions of the system be monitored. This monitoring must not, of course, alter the components as designed, or the data collected would be invalid. This results in the need for instrumented connectors.

WRIGHT can represent these easily by adding a new Listener role to each connector specification and altering the **Glue** to copy data to the new participant. For example, an instrumented ClientServer connector could be described as in figure 4.14. The same Listener can also be added to any of the other connectors. For example, InstrumentedDClientServerPush is shown in figure 4.15.

This method of adding a Listener role to each connector type both supports the need for instrumentation and retains the simplicity of the un-instrumented architecture. By altering the connector specifications, we indicate the need for this functionality in the system and provide a basis for analyzing alternatives for how it is to be supplied. Perhaps a single, centralized monitoring database component is best, or perhaps a distributed approach, where there is a monitor for each component, is preferable. By including the specification of the

Configuration Testbed2
[..Type definitions..]
Instances
ExperimentControl : ThreadedMixedComp (1,0,0,0,0,0,0,2)
DoctrineAuthoring : DynamicServerized(1,3)
DoctrineValidation : DoctrineValidationT
TrackServer : ThreadedMixedComp (3,0,1,0,1,0,0,0)
GeoServer : GeoServerT
DoctrineReasoning : DoctrineReasoningT
DisplayServer : DisplayServerT
$CS_{1..4}$: ClientServer
$DCS_{1..5}$: DClientServer
$DCSPush_{1..4}$: DClientServerPush
OpenLoop : OpenLoopBuffer
Attachments
ExperimentControl.DPushClient **as** $DCSPush_1$.Client
DoctrineAuthoring.Service **as** $DCSPush_1$.Server
ExperimentControl.DPushClient **as** $DCSPush_2$.Client
DoctrineValidation.ExCtrl **as** $DCSPush_2$.Server
ExperimentControl.PushServer **as** CS_1.Server
TrackServer.PullClient **as** CS_1.Client
DoctrineAuthoring.Client **as** DCS_1.Server
DoctrineValidation.DoctAuth **as** DCS_1.Client
TrackServer.PushServer **as** DCS_2.Server
DoctrineValidation.TrSrv **as** DCS_2.Client
DoctrineAuthoring.Client **as** DCS_3.Server
DoctrineReasoning.DoctAuth **as** DCS_3.Client
DoctrineAuthoring.Client **as** DCS_4.Server
GeoServer.DoctAuth **as** DCS_4.Client
TrackServer.PushServer **as** CS_2.Server
DoctrineReasoning.TrSrv **as** CS_2.Client
TrackServer.PushServer **as** CS_3.Server
GeoServer.TrSrv **as** CS_3.Client
GeoServer.DoctReas **as** OpenLoop.Source
DoctrineReasoning.GeoSrv **as** OpenLoop.Target
DoctrineAuthoring.Client **as** DCS_5.Server
DisplayServer.DoctAuth **as** DCS_5.Client
TrackServer.PushServer **as** CS_4.Server
DisplayServer.TrSrv **as** CS_4.Client
DisplayServer.DoctVal **as** $DCSPush_3$.Server
DoctrineValidation.DispSrv **as** $DCSPush_3$.Client
DisplayServer.DoctReas **as** $DCSPush_4$.Server
DoctrineReasoning.DispSrv **as** $DCSPush_4$.Client
End Testbed2.

Figure 4.12: Revised AEGIS configuration.

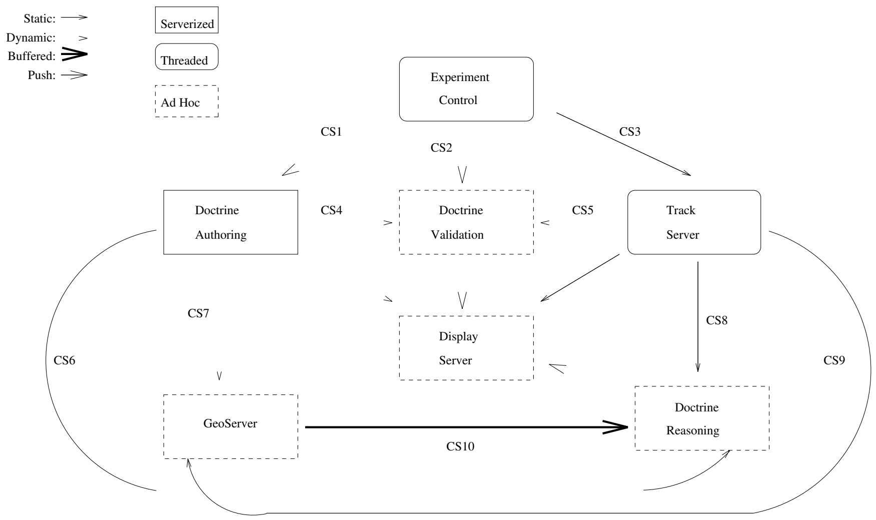

Figure 4.13: Actual AEGIS prototype architecture.

Connector InstrumentedClientServer =
Role Client = ClientPullT
Role Server = ServerPushT
Role Listener = data?x →Listener [] §
Glue = Client.open →$\overline{\text{Server.open}}$→**Glue**
[] Client.close→$\overline{\text{Server.close}}$→**Glue**
[] Client.request→$\overline{\text{Server.request}}$→**Glue**
[] Server.result?x→$\overline{\text{Client.result!x}}$
→$\overline{\text{Listener.data!x}}$→**Glue**
[] §

Figure 4.14: An instrumented connector.

Connector InstrumentedDClientServerPush =
Role Client = DClientPushT
Role Server = DServerPullT
Role Listener = data?x→Listener [] §
Glue = Client.open →$\overline{\text{Server.open}}$ →**Glue**
[] Client.request?x →$\overline{\text{Server.request!x}}$
→$\overline{\text{Listener.data!x}}$ →**Glue**
[] Server.result →$\overline{\text{Client.result}}$ →**Glue**
[] Client.close →$\overline{\text{Server.close}}$ →**Glue**
[] §

Figure 4.15: Another instrumented connector.

Listener in the architecture, we ensure that this concern will be addressed by all parts of the implementation, avoiding inconsistencies where some data is collected and other data not, perhaps presenting unrealistic or even misleading information to analysts.

At the same time, the basic client-server interaction remains clear, and the core component specifications are unchanged (there will, of course, be additional monitoring components that are introduced). Thus, WRIGHT provides an effective means of introducing concerns at the architectural level, precisely defining alternative solutions, and analyzing those solutions to determine which is best for a given situation.

4.5 Discussion

In this chapter we have seen, through the example of the AEGIS prototype architecture, that WRIGHT can be used to provide a formal specification of a non-trivial architectural configuration. As we have illustrated, this specification illuminates many of the issues left unresolved in a less formal treatment.

The primary benefit of this specification has been its precision and its attention to detail. We note, however, that although we have been quite specific about the protocols of interaction, the specification is considerably more abstract than the actual functional behavior of the components in the system. Thus, the specification provides a basis for reasoning about the system and at the same time may be used as a basis for more concise descriptions of the functioning system. In effect, the WRIGHT specification gives meaning to informal descriptions such as shown in figure 4.1.

In the formalization of AEGIS, we have introduced a number of techniques for the abstract description of software architecture, including:

- **Separating Interaction Patterns:** In describing our connectors, we were careful to keep separate interactions separate. Both the client-server interaction pattern described by the ClientServer connector and the buffered data interaction described by OpenLoopBuffer are implemented via a single operating system library, `stdio`. However, they represent different abstract interaction patterns, and so we keep them separate. By recognizing the different usage patterns, we can consider the different effects that a single infrastructure platform can have on the computation, depending on its use.

- **Separating Interaction from Computation:** WRIGHT promotes the separation of component descriptions into interface and computation, and of configurations into connectors and components. By distinguishing the servers abstract behavior *at a single interface point* from its overall behavior, we can raise the level of abstraction of the system, showing how it matches the informal "client-server" characterization while at the same time discovering properties based on the specifics of how an individual computation combines its interfaces. By separating, for example, the ClientPullT and ServerPushT interface types from the Client and Server specifications, we can show how MixedComp truly *is* a combination of the client and server computations, and be sure that it can be used in any context where either is expected. Had we not made this separation, the relation between the component types would have been much more difficult to establish.

- **Focussing on Patterns of Control:** In each part of the AEGIS specification, we described an abstract pattern of control. The very first specification, of the ClientPullT interface type, shows that a client in the client-server interaction repeatedly *decides* whether to make a request or to close the connection. The ClientServer connector type indicates that all control mechanisms are located in the components, while the OpenLoopBuffer provides additional control mechanisms through its buffer. The Server component type indicates that, surprisingly, it requires control over observed events. This latter observation is really only possible because of careful adherence to this technique: At each point in a computation or an interaction protocol where there is more than one possible action, it must be determined whether a given party (such as component) *decides* what to do or *accepts* the occurrence of either possibility. If the component decides, it is an internal choice, and is represented via $\sqcap$. If the component has no influence on what can happen *and* is capable of dealing with either possibility, then an external choice is made, via $\Box$.

- **Using Parameters to Generalize and Abstract:** Another important aspect of a focus on patterns of control is that what we describe are patterns, not specifics. We chose to describe a Server component type, showing a general pattern of responding to clients' requests, rather than to describe the detailed computations of the TrackServer, GeoServer, *etc.* By recognizing the commonality and by parameterizing our descriptions of the component types, we were able to raise the level of abstraction of our description and increase our ability to consider general solutions to the problems with the initial architecture. This approach is most obvious in the ThreadedMixedComp, in which we capture a fully general threaded client-server.

- **Showing All Interfaces:** A critical aspect of software architectural description is that all interactions in a system be described. If we hope to ensure the consistency of the system and to be able to reason abstractly about alternatives, we must be sure that there are no hidden dependencies between elements that are abstractly independent. Thus, when we describe what is abstractly a "server" in the architecture, we must also consider whether it acts as a "client" to other components. Similarly, if monitoring of interactions is a key function of the system, we must provide interfaces for that, as new roles in the connectors, so that we can describe and analyze the alternative approaches to that aspect of the system. If we omit an interface between components that do interact, we lose our ability to reason about the behavior of those components independently and to consider changes to that interaction.

- **Separating Different Roles:** When we consider an issue such as instrumentation at the architectural level, we have the ability to separate it from the details of core functionality and implementation. We show this separation by providing a separate role for the Listener from the Clients and Servers. One possible implementation of instrumentation is to have each component record its own outgoing messages in a log. This could have been described by including a logging function in each component type. But this would have restricted our ability to analyze the system in two ways: First, it describes only one possible solution to the monitoring problem, and removes our ability to consider alternatives at the architectural level. Second, it fails to separate the abstract issue of computation of data values and their communication and the issue of monitoring in an

> experimental system. Combining the two would have obscured the client-server relation by adding additional details in the middle of that part of the computation, while also obscuring the monitoring function by distributing it across the entire computation. If we do end up constructing a combined implementation, that can be represented in our separated model by adding a listener port to each component type and attaching the listener roles to the appropriate components. We still retain the abstract separation of these issues while correctly describing the chosen solution.

Although we have concentrated on describing AEGIS as a single configuration, our formalization of interface, component, and connector types would permit us to reason about an architectural style based around this vocabulary. In effect, the AEGIS developers' efforts at architectural design were based on an informal understanding of a "client-server" style. They decided on the style and then used infrastructure, Unix sockets, that they believed to support the style. Had there been a formal specification of clients, servers, and client-server connectors, such as we have shown in this chapter, it is possible that they would not have made the choices they did, which resulted in a complex, ad hoc solution. Style constraints could have indicated, for example, that if static client-server connectors are to be used with single-threaded servers, no two clients may make requests to the same two servers. This constraint would have informed the AEGIS developers that the choices they made, based on their informal understanding, were not suitable, and led them to a simpler, more successful solution.

We will see in the next chapter how arguments about, for example, absence of deadlock and substitutability of one connector or component type for another can be made in a rigorous fashion. In the context of an architectural style, these results become general rules that can be applied to all instances of the style. Hence the architectural level specification becomes cost effective through amortization of its results across a wide variety of systems.

Chapter 5

Case Study: Justo-Cunha Style

5.1 Introduction

In this chapter we shift our attention from the architecture of individual systems to consider families of systems that share common characteristics. These families of systems are architectural styles. By considering styles, the developer shifts his or her attention and analysis from the task of locating potential problems in the architecture of single systems to reasoning about the properties that are shared by all configurations in a style. The designer of a style hopes to reduce the effort of system developers by carrying out important analysis in general, so it can be applied directly to systems in the style. That is, by establishing that a system's architecture is in a particular style, the system developer can be sure that the system will have all of the properties that have been demonstrated for the style.

For example, in our case study we will show that all systems in the style are globally deadlock-free (*i.e.*, that they will never stop sending messages because every component is waiting for some other component). In general, global deadlock-freedom is a difficult property to establish about a system, requiring exhaustive consideration of all states of the system. Establishing that a system is in our example style, on the other hand, is relatively easy: there must be no cycles in the architecture's configuration. Thus, a system developer can reduce his or her effort by using the style: the difficult deadlock test is replaced by the simple style membership test.

A typical informally-described style consists of a set of connectors with a rough description of how they behave, some examples of components that might appear in the style, again with some description of the behavior, some rules of thumb for combining the components and connectors into a configuration that fits the style, and a description of the desirable properties that systems will have if the style rules are properly followed. The justification of the style by its properties (an informal proof) also serves as a guide to the developer in trying to figure out more subtle points of system construction.

For example, our case study is based on a style originally described by Justo and Cunha [JC94]. The purpose of the style is to support the development of message-passing systems that are known to be deadlock-free. They provide a semi-formal description of the style using informal connectors, code templates for components, and English language constraints on how configurations can be constructed. Justo and Cunha provide informal proofs that, when these constraints are obeyed, the constructed systems will be globally deadlock-free.

While use of these informal styles can save developers considerable effort both in the design and analysis of their systems, particularly if the developer is experienced in the use of the style, there are several weaknesses to this informal approach. First, it is not clear exactly where the boundaries of the style's family of systems is. That is, there will be systems that cannot be shown either to fit the style, or not to fit the style. This means that the developer of that system does not know whether the analyses performed for the style apply or not. Further, because the justifications are informal, it may be the case that they contain exceptions or errors that are not made obvious in the informal description. Also, because the reasoning behind the justification and analyses may not be spelled out, it will be impossible to know whether a weaker form of the properties applies if the exact constraints of the style are not met, or whether they have been completely voided: The analysis must be redone.

By formalizing a style in WRIGHT, we provide important benefits to the software architect that are not achieved by this informal technique. First, we are able to be much more precise in understanding what it is that the style actually covers. That is, the family of systems represented by the style is delineated precisely. Second, we provide a rigorous basis for analysis of the style's properties. The informal justifications can be made into formal guarantees that critical properties hold, and the rule-of-thumb tests are turned into precisely bounded constraints. Third, by exploring the precise boundaries of the constraints imposed by the rules of thumb, we can generalize the property justifications, thus establishing the critical guarantees over a wider range of systems. This makes it easier for the developer to build a system that serves his purpose while still using the style constraints to simplify the analysis task.

Our case study will begin with the introduction of vocabulary for the description of configurations. To create an initial vocabulary, we will simply formalize the description provided in [JC94]. We will use this as a starting point to explore how we might formalize the informal constraints described by Justo and Cunha. Through this attempt we will see that the informal descriptions are in fact ambiguous: Depending on how they are considered, they don't guarantee deadlock-freedom at all! We therefore realize that the constraints must be considered in a particular way in order to guarantee the desired property.

Given the new, more restrictive constraints, we will show how the property of global deadlock-freedom can then be formally proven to hold for any instance of the JC style. Using the architectural structure provided by the style as a guide, we are able to apply the standard technique of inductive reasoning over the traces of a CSP process. The architectural structure reduces the complexity of the proof, making it tractable. We thus justify the use of the vocabulary and constraints imposed by the style, by showing how they make it easier to establish an important semantic property of the systems constructed. This property is established for every configuration that obeys the style constraint; the developer knows that his or her system is deadlock-free without repeating the proof.

By formalizing the informal description, however, we realize that the proofs as specified only apply over a fairly restricted set of systems. While the vocabulary seemed flexible at first, and the constraints reasonable, the formalization reveals that the style as originally specified didn't cover many systems that we would like to construct, and which we believe to have the property of interest: global deadlock-freedom. We will capitalize on the insights from the initial formalization to further generalize the style, extending the deadlock-freedom result to cover a wider family of systems.

To prove global deadlock-freedom in the more general style, we introduce the technique of *encapsulation*. In this technique, we demonstrate that the behavior of any configuration in the style is equivalent to the behavior of some individual component. Thus, any configuration can be encapsulated (have an interface boundary drawn around it) and treated as a component in another, larger configuration in the same style. The way we will use this in our proof of deadlock freedom is that, if we can show that all single components are deadlock-free, and every configuration can be viewed as a component, then all configurations must be deadlock-free. The encapsulation result is a powerful technique for analysis of architectural styles, because it means that *any* property that is guaranteed for the component types is automatically extended to all configurations. Thus, by establishing that the (relatively simple) style constraints are met by a system, the software developer can immediately use any result that has been proven for the component types.

Further, the encapsulation property provides a justification for the hierarchical structuring of the system architecture during design. By permitting the developer to hide details of the structure at each level of consideration, hierarchical decomposition can provide important benefits in terms of simplifying exposition of the system and raising the level of abstraction at which it is described. It simplifies exposition by reducing the complexity of each level in the hierarchy and making it clear where subsystems are independent, and it raises the level of abstraction by hiding the details of how subsystems are constructed.

Finally we will step back and summarize the key ideas and techniques introduced in this chapter, showing how the ideas generalize from the case study to the more general problem of style specification and analysis.

5.2 The Justo-Cunha Style

In the case study described in this chapter, we formalize and analyze the architectural style informally described by Justo and Cunha in [JC94]. This style, which we term the Justo-Cunha, or JC, style, is designed to simplify the determination of global-deadlock freedom in distributed message-passing systems. Buffered message-passing is a common way of configuring systems [BWW88, Kra90]. An important problem that arises in these systems is global deadlock of systems. Deadlock arises because components must wait for messages from other components before they may proceed with their own computation and output messages. If it occurs that all processors are waiting for messages and none are capable of outputting messages, then the system deadlocks.

Justo and Cunha describe their style as a single connector type, a pair of component templates, and a set of configuration rules. The connector type is a message buffer. Justo and Cunha describe it as follows:

> The send primitive in CL does not block the sender. The sent messages are usually stored in a circular buffer. If the buffer is full, then the first messages received in the buffer are overwritten. The size of the buffer can be specified by the user [JC94, p. 149]

```
task <name.SR> (entryport in:ArrayOfPorts1;
               exitport out:ArrayOfPorts2)
    <declarations>
  begin
    <initialization>
    repeat
      send message[1] to out[1];
        ...
      send message[m] to out[m];
      <compute>
      receive message[1] from in[1];
        ...
      receive message[n] from in[n];
      <compute>
    until true
  end <name.SR>.
```

Figure 5.1: A component template from the JC Style.

All components in the style represent processes that send and receive messages via message buffers. Justo and Cunha define the set of permitted components by providing two component templates that developers of systems in the style must copy and then modify in specific ways. An example of such a template is shown in figure 5.1. Justo and Cunha describe them as follows:

> The `S-R` template can send its messages and then it may block waiting to receive its messages. The ⟨*compute*⟩ part correspond [sic] to a block of statements that should not include communication and should terminate...
>
> [The `R-S`] template is the dual of the `S-R` template. In this case, the template first receives its message, performs some computation and then sends messages. [JC94, p. 149]

In order to use these templates effectively, Justo and Cunha provide three configuration rules [JC94, p. 152]:

Rule 1 A cycle of R-S template processes always deadlocks.

Rule 2 A cycle of S-R template processes never deadlocks.

Rule 3 A cycle of R-S and S-R template processes never deadlocks.

Intuitively, we can see that these rules hold because deadlock can occur only when all components are waiting for a message. While R-S components may block immediately, causing a potential deadlock, S-R components begin by sending a message. These messages then travel through the system, ensuring that progress is always possible somewhere in the system.

While the application of these rules is clear in the simple case of systems consisting of exactly a single cycle of components, it is not as clear how or whether these rules can be applied to more complex system configurations. For example, does this rule mean that a cycle of R-S components will cause deadlock regardless of the context in which it appears?

Thus, we can see a number of weaknesses in this informal exposition of an architectural style. The informal configuration rules are not sufficiently specific about what systems are or are not included in the style. We don't know whether there are exceptions to the rules, so that some more complicated systems might deadlock even though they seem to obey the rules. The coding templates provide examples of permitted components, but there might be other, more flexible patterns that could be used and still retain the property of deadlock freedom. For example, the JC style as described doesn't seem to permit a component to vary the order in which the messages are sent. Is this a critical property of the style, or only coincidental to the way that the templates are written? The informal description does not clarify this point.

5.3 A Formal Description of the JC Style

We now show how the informal description of the JC style provided by Justo and Cunha can be formalized using WRIGHT. Our initial formalization will simply make precise the semantics of the templates and connector description provided in [JC94]. This formalization, defined as WRIGHT component and connector types, will serve as a starting point to explore how we might formalize the informal constraints described in [JC94]. Through this exercise will show that the informal descriptions are in fact ambiguous: Certain reasonable interpretations of the constraints are not correct; if a developer simply reads the rules and follows them without mentally reconstructing the proof of global deadlock-freedom for his particular system, he can't be sure that his system will operate without deadlock. In effect, the rules could actually result in *more* trouble for the developer: if they are interpreted in ways that Justo and Cunha didn't expect, then a system may in fact deadlock, while the developer believes that it doesn't, because of the "guarantees" provided by use of the style.

By formalizing the constraints, we will see under what situations they do or don't apply, and will show how, given a restrictive form of the constraints, the property of global deadlock-freedom can be formally proven to hold for any instance of the JC style. Using the semantics of the component and connector types, and the formal style constraints as a guide, we apply the standard technique of inductive reasoning over the traces of a CSP process. The architectural structure reduces the complexity of the proof, making it tractable. We thus justify the use of the vocabulary and constraints imposed by the style: They make it easier to establish an important semantic property of the systems constructed. The developer who uses the formalized JC style knows that his or her system is deadlock-free without repeating the proof, because we have established this property for every configuration that obeys the style constraint.

5.3.1 Interface Types

The first element of a style description is the introduction of interface types. There are two interfaces in the JC style: Receiver and Sender:

Interface Type $\mathsf{ReceiverType} = \overline{\mathsf{receive}} \rightarrow \mathsf{ReceiverType}$

Connector Message(n:1..)
  Role Sender = SenderType
  Role Receiver = ReceiverType
  glue = buf_0 **where**

$$\text{buf}_i = \begin{cases} \text{Sender.send} \rightarrow \text{buf}_1, & \textbf{when } i = 0 \\ \text{Sender.send} \rightarrow \text{buf}_n \,[]\, \text{Receiver.receive} \rightarrow \text{buf}_{n-1}, & \textbf{when } i = n \\ \text{Sender.send} \rightarrow \text{buf}_{i+1} \,[]\, \text{Receiver.receive} \rightarrow \text{buf}_{i-1}, & \textbf{otherwise} \end{cases}$$

Figure 5.2: The Message buffer connector in WRIGHT.

Interface Type SenderType = $\overline{\text{send}} \rightarrow$ SenderType

Because Justo and Cunha only consider non-terminating computations, the interfaces shown are simple repetitions of a single event. In general, however, an interface type might show how termination is achieved, and whether users of that interface control the decision about termination. For example, if we wanted to indicate that the `SenderType` is not obligated to send an infinite number of times, we would add a *choice* of termination:

Interface Type AlternativeSenderType = $\overline{\text{send}} \rightarrow$ AlternativeSenderType $\sqcap$ §

5.3.2 Connector Types

The second element of a style description is the introduction of *connector types*. These types describe the kinds of interactions that can occur in systems developed using the style.

Justo and Cunha describe the interactions of their style informally as follows: ".. the send primitive in CL does not block the sender. The sent messages are usually stored in a circular buffer. If the buffer is full, then the first messages received are overwritten. The size of the buffer can be specified by the user."

To formalize this interaction WRIGHT provides the connector type. A connector type specification is divided into three parts, each of which gives different information about the interaction. The connector type of the Justo-Cunha style, the Message connector, is shown in figure 5.2.

In the Message connector we see how the connector uses the signature to provide a single dimension of variability, as specified by Justo and Cunha: the size of the finite buffer. The buffer size is indicated by the parameter n in the connector specification.

The second part of the specification, the *role* specifications, indicates the participants in the interaction and the constraints on those participants. These use the interface types defined above.

The third part of the connector specification is the *glue* specification, which indicates how the participants are combined to create an interaction. In the message connector, the interaction is mediated by a circular buffer that permits a single receive for each send, unless the buffer is full, in which case it drops any further sent messages until a receive releases space in the buffer.

Component S-R(r:1..;s:1..)
  Port $\mathsf{In}_{1..r}$ = ReceiverType
  Port $\mathsf{Out}_{1..s}$ = SenderType
  Computation = S_1 **where**

$$\mathsf{S}_i = \begin{cases} \overline{\mathsf{Out}_i.\mathsf{send}} \rightarrow \mathsf{S}_{i+1}, & \\ \qquad \textbf{when } 1 \leq i \leq s & \\ \overline{\mathsf{In}_{i-s}.\mathsf{receive}} \rightarrow \mathsf{S}_{i+1}, & \\ \qquad \textbf{when } s < i < r+s & \\ \overline{\mathsf{In}_r.\mathsf{receive}} \rightarrow \mathsf{S}_1, & \\ \qquad \textbf{when } i = s+r & \end{cases}$$

Component R-S(r:1..;s:1..)
  Port $\mathsf{In}_{1..r}$ = ReceiverType
  Port $\mathsf{Out}_{1..s}$ = SenderType
  Computation = R_1 **where**

$$\mathsf{R}_i = \begin{cases} \overline{\mathsf{In}_i.\mathsf{receive}} \rightarrow \mathsf{R}_{i+1}, & \\ \qquad \textbf{when } 1 \leq i \leq r & \\ \overline{\mathsf{Out}_{i-r}.\mathsf{send}} \rightarrow \mathsf{R}_{i+1}, & \\ \qquad \textbf{when } r < i < r+s & \\ \overline{\mathsf{Out}_s.\mathsf{send}} \rightarrow \mathsf{R}_1, & \\ \qquad \textbf{when } i = s+r & \end{cases}$$

Figure 5.3: S-R and R-S templates.

Consistency of the Message Connector

As in our previous case study, we must apply WRIGHT's checks for consistency before we can use the connector. In this case, the simplicity of the interfaces makes it easy to demonstrate that the connector obeys the rules for connectors. The roles are clearly deadlock-free, since they never terminate and have only one event (test 3). The connector is deadlock free because it never refuses the send event (test 2). receive is initiated only by Receiver and send only by Sender, so there is a single initiator for each event (test 4). The **Glue** has no initiated events, and each role has only one, so initiator commits is obeyed (test 5).

5.3.3 Component Types

Beyond the description of the interactions in the JC style, Justo and Cunha provide two component templates to be used. These correspond to the third element of a WRIGHT style description, the component types. The WRIGHT formalization of the JC templates, termed S-R and R-S templates, are shown in figure 5.3.

In the R-S and S-R templates, the number of inputs and outputs can be varied, as indicated by the parameters r and s. In the J-C message passing style, an In port will fill a Receiver role and an Out port will fill a Sender role. Thus in this example, they use the same interface types as the corresponding role. In general, as we discussed in chapter 3, they need not.

In a component specification, the **Computation** indicates how the component's interactions are combined to form a distinct computation. It is in the **Computation** that the S-R and R-S component templates differ. The S-R template begins by sending messages on each of its Out ports and then waits to receive messages on each of its In ports. The R-S template reverses this pattern of events, first waiting for messages and then sending. Because we are only interested in the *pattern* of messages, and because the JC component templates do not permit components to alter the ordering of events, we have not shown the data that is transmitted with the send and receive events. Of course, any implementation of a system using this style would carry data with the messages, and instances of these templates would carry out different calculations in order to provide specific data values. The purpose of the formalization of this style is to analyze systems for global deadlock-freedom, and so we abstract away from the parts of the computation (namely, the computation of data) that do not influence that issue. It is the responsibility of the developer using the style to show that any concrete components obey the abstract pattern of messages described by these component types. If the code templates provided by Justo and Cunha are used directly, this follows in a straightforward manner.

Consistency of the Component Types

As with the connector type, we must confirm that the component types pass the WRIGHT checks before we can use them. The tests that apply to component types are port/computation consistency (test 1) and initiator commits (test 5). Each of the computations is capable of exactly one event at a time, so the computation commits to that event: Initiator commits is clearly obeyed. Further, the computation cycles through each event before repeating another, and the computation does not terminate, so each event is produced an infinite number of times. This is exactly the pattern of events represented by the ports, so port-computation consistency is also obeyed.

5.3.4 An Initial Analysis

As we saw in the previous case study, the introduction of vocabulary is sufficient to construct architectural configurations. We could stop here, and already have provided an aid to developers who wish to formalize and analyze their systems, by making it easier for them to describe the semantics of their systems through the use of our pre-defined vocabulary.

However, in this case, as we usually will, we want to take the style further. We want to provide both guidance in how the vocabulary should be used and perform analyses that will benefit potential users of the style. We need to indicate such things as: Does the style permit the use of additional vocabulary, or are these types to be used exclusively? Are there specific component or connector instances that must appear in every configuration, or is the developer free to introduce or exclude any given element? Can elements be combined in any way that the developer sees fit, or are there rules on the topology of attachments that must be obeyed?

Some of these constraints can be determined directly to reflect the intent of the style designer, but others are better considered in terms of potential analysis of the style. In our case study, we are concerned with questions of global-deadlock freedom, and so we will want to use constraints that ensure this property, excluding any configurations from the style that have the potential to deadlock, and proving that any system that obeys the style's constraints must be deadlock free.

The first constraint for the JC style is a common one: Justo and Cunha only consider systems in which all components and connectors use the vocabulary that they introduce. This kind of constraint makes sense, because the introduction of new vocabulary means that the semantics of those elements is unconstrained, and it is difficult, if not impossible, to analyze systems with unconstrained semantics as a class. The first constraints, which we will refer to as $JC\_Constraints\_1\,a$, are:

$$\begin{aligned} &\forall\, c : Component \bullet Type(c) = \textsf{S-R} \vee Type(c) = \textsf{R-S} \\ \wedge\ &\forall\, c : Connector \bullet Type(c) = \textsf{Message} \end{aligned}$$

These are constraints that help to define the style. By placing these in the **Constraints** section of a style definition, we indicate that every system that is in the JC style must obey these constraints. If an architecture does not obey these constraints, they are not in the JC style, *by definition*.

This contrasts with another property that we want to consider, deadlock-freedom. We will define deadlock-freedom as a predicate that appears similar to the constraints of $JC\_Constraints\_1\,a$, and in fact the semantics of the predicate are defined in the same way, by evaluating a given system architecture. However, we don't want to apply deadlock-freedom as part of the definition of the style, because this wouldn't simplify the task of a system developer. By making deadlock-freedom a constraint of the style, we make it a *proof-obligation* on any user of the style. That is, before we can do anything with a system based on it being in the style, the developer must prove that it is deadlock-free. Instead, we want to provide deadlock-freedom as a *style property*. That is, the developer uses other constraints that define the style (such as $JC\_Constraints\_1\,a$) to prove that his system is in the style. Then, because of the properties of the style, he knows, *without further analysis by the developer*, that the system must therefore be deadlock-free.

We will now define the property of deadlock-freedom, and then show how different constraints may be defined as part of a potential JC style. We will use these constraints to analyze the style and show that some of the sets of constraints guarantee deadlock freedom and some do not.

Recall that deadlock is a situation where a system can neither make progress nor terminate successfully. In order to formalize this property, we must refer to the overall behavior of the architecture, which is the CSP process $System$ (as defined in chapter 3). We can use the traces and refusals of $System$ to define global deadlock and deadlock-freedom formally as follows:

$$\begin{aligned}
&Deadlock \mathrel{\widehat{=}} \exists\, t : Traces(System) \\
&\qquad \mid (t, \alpha System) \in Failures(System) \wedge last(t) \neq \surd \\
&Deadlock_Free \mathrel{\widehat{=}} \neg\ Deadlock
\end{aligned}$$

As for our definitions of local deadlock in chapter 3, this indicates that deadlock occurs when the system cannot make progress (it refuses every event, in the first half of the condition) and has not terminated successfully (the final event is not $\surd$, which represents success). Any process that cannot deadlock is deadlock-free.

It is easy to see why it is not useful to simply apply $Deadlock\_Free$ as a constraint: By relying on the unstructured semantics of the system to define the property, the predicate provides little or no guidance on how it may be achieved by an actual system architecture, and it is not clear how this property can be established without exhaustive analysis of the detailed semantics of a system. It is exactly to make the proof of this property *easier* that Justo and Cunha set out to define their style. To include it as a proof obligation before the style's analysis can begin defeats the purpose of the exercise.

We therefore turn to the constraints that Justo and Cunha define. The idea of these constraints is that (a) they will be easy to establish for a given configuration, and (b) we can prove that if they are established, deadlock-freedom is guaranteed. This will make the job of the system developer much easier, because our analysis establishes (b) once and for all: the difficult deadlock-freedom proof is carried out once, proving a theorem of the form

Figure 5.4: Two configurations: cycle, non-cycle.

"$JC\_Constraints \Rightarrow Deadlock\_Free$." In order to gain the benefits of deadlock-freedom for any system, only the easier-to-demonstrate style constraints must be shown.

5.3.4.1 Basis for JC Constraints: Dependency and Cycles

Justo and Cunha's configuration rules are all based on the idea of cycles of components. Their idea is that the graph of connectors attached to components defines dependencies between components, and that any deadlock that occurs will be because of a cycle of dependencies.

But what exactly constitutes a "cycle" of components? Already we can see that the informality of the original style definition presents problems. Consider two systems, shown in figure 5.4. Both have two connectors between two components, but from the way the connectors are drawn, we deduce that the first has a cycle and the other does not.

The difference in the drawings is in the arrows: The arrowheads indicate that the connectors have a "direction." In the JC Style, the direction of the connectors that is used in determining cycles is based on the roles. Connector arrows are drawn from the Sender role to the Receiver role. We represent this as a relation between components, $ConnectsTo$:

$$
\begin{aligned}
&\forall\, c_1, c_2 : Components \bullet ConnectsTo(c_1, c_2) \mathrel{\widehat=} \\
&\quad \exists\, conn : Connectors;\ p_1 : Ports(c_1);\ p_2 : Ports(p_2) \mid \\
&\qquad \{((c_1, p_1), (conn, \textsf{Sender})), ((c_2, p_2), (conn, \textsf{Receiver}))\} \subseteq attachments
\end{aligned}
$$

This predicate indicates that a component c_1 "$ConnectsTo$" another component c_2 if they share a connector such that c_1 plays the Sender role and c_2 plays the receiver role in the connector. The reason that we use this particular definition of $ConnectsTo$ (rather than a non-directed version, or one where the roles are reversed) will become clear when we start to analyze the system for deadlock.

With this notion of directedness, we can now define a cycle of components. A set of components is a cycle if they form a sequence of components where each component is connected to the next in the sequence, and the last connects to the first:

$$
\begin{aligned}
&\forall\, C : \mathbb{P}\, Components \bullet Cycle(C) \mathrel{\widehat=} \\
&\quad \exists\, S : \mathrm{seq}\, C \mid \mathrm{ran}\, S = C \bullet \\
&\qquad\quad \forall\, i : 1..\#S - 1 \bullet ConnectsTo(S(i), S(i+1)) \\
&\qquad \wedge\, ConnectsTo(S(\#S), S(1))
\end{aligned}
$$

5.3.4.2 Justo and Cunha's Configuration Rules

Now that we have the concepts of a cycle and of deadlock-freedom formalized, we can consider how to formalize the three configuration rules presented by Justo and Cunha. Recall their statement by Justo and Cunha:

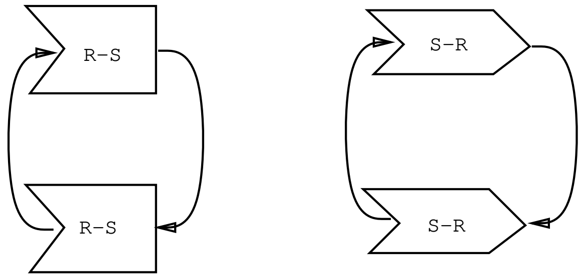

Figure 5.5: Counter example to deadlock claim, rule 1.

1. A cycle of R-S template processes always deadlocks.
2. A cycle of S-R template processes never deadlocks.
3. A cycle of R-S and S-R template processes never deadlocks.

As we can see, these "rules" are not actually constraints on systems, but rather theorems about particular topologies constructed in the JC style. While these rules are reasonable as informal design guidelines, they illustrate some of the ambiguities that must be resolved when formalizing an informal description. For example, we must ask ourselves in what situations the rules apply. How can we use these results on cycles to constrain our construction of entire systems? Consider the following (perhaps overly naive) potential rephrasing of the first theorem:

Any system containing a cycle of R-S components will eventually deadlock.

This statement can be formalized as follows:[1]

$$(\exists\, C : \mathbb{P}\, Components \mid Cycle(C) \wedge \forall\, c : C \bullet\, Type(c) == \textsf{R-S}) \Rightarrow Deadlock$$

This interpretation of the rule turns out to be incorrect, as the system in figure 5.5 shows. This system will continue to make progress (in the S-R cycle) even though the R-S cycle never executes. This observation leads us to a more accurate interpretation of rule 1:

No component in a cycle of R-S components will ever send.

This is formalized as ($JC\_Theorem\_1\,a$):

$$\begin{array}{l}\forall\, C : \mathbb{P}\, Components \mid Cycle(C) \wedge (\forall\, c : C \bullet\, Type(c) = \textsf{R-S}) \\ \quad \bullet\, \forall\, t : Traces(System);\, c : C \bullet\, last(t) \neq Name(c).\textsf{send}\end{array}$$

This rule is now provable: Consider the behavior of an R-S component, which we will call r. In order to send, r must first carry out a receive on each of its Receive ports. Because there is a cycle, one of these ports is attached to a Message connector as the Receiver role. Initially, the Receiver.receive event is refused — it must be preceded by a send in the Sender role. But, again because of the existence of the cycle, we know that

[1]In this section, all of the theorems assume that *JC_Constraints_1a* hold, since these will be part of the eventual complete style constraints.

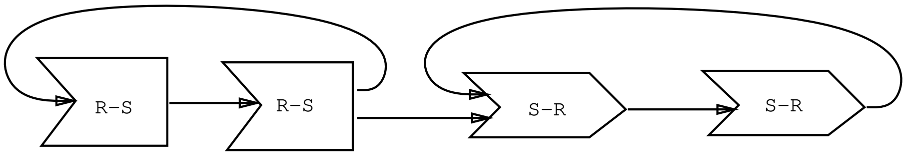

Figure 5.6: Counterexample to rule 2.

the Sender component of this connector, s, is also in the cycle. Therefore, because s must send before r finishes receiving, r cannot be the first component in the cycle to send. But this argument applies for *any* component in the cycle, so none of them can be first to send. Therefore no component in the cycle will ever send.

As with rule 1, the informality of the other rules present difficulties when we attempt to interpret them more precisely. If we assume that the rules are intended in their fullest generality, that they indicate that an S-R cycle will never deadlock regardless of the context in which it appears, we again discover a weakness in the rule. Figure 5.6 shows a system where the additional presence of an R-S cycle will cause the system as a whole to deadlock, even though there is an S-R cycle present. In this system, each of the S-R components will send once, but then the left-hand S-R component will block forever waiting for a send from the R-S cycle (which we know from $JC\_Theorem\_1a$ will never happen).

Again we can ask if there is a reasonable restriction of rules 2 and 3 that is correct. One such restriction is the following:

A system consisting only *of a single cycle of S-R and R-S components will not deadlock.*

This is formalized as ($JC\_Theorem\_2a$):

$$\begin{aligned}
&\quad Cycle(Components) \wedge \#Connectors = \#Components \\
&\wedge\ \exists\, c : Components \mid Type(c) = \textsf{S-R} \\
&\qquad \Rightarrow Deadlock_Free
\end{aligned}$$

In this predicate $Cycle(Components)$ indicates that there is a cycle consisting of all of the components. $\#Connectors = \#Components$ together with the cycle property ensures that there are no connections except those participating in the cycle, and the existence of at least one S-R component guarantees that there is a component that will introduce an initial message into the system, after which progress can always be made.

Briefly, this property can be proved by induction on the traces of the system. Each S-R component initially introduces an active message into the system. Every subsequent event is either a send event, moving a message into a message buffer, or a receive event, activating a blocked component. The fact that the system consists of exactly a cycle ensures that each component has exactly one input that comes from within the system, and hence that whenever it receives on that input port it is subsequently able to send an output before it is blocked again. The only way messages can be lost is by having a full message buffer. This can never reduce the number of messages to zero. Since progress is always possible whenever there is at least one message active in the system, deadlock cannot occur. A more formal proof follows.

Proof: By induction on the length of a trace.

Let $Prog(P)$ = the number of components that can send ($Prog_s$) + the number of messages pending in the system ($Prog_m$).

We will show that if $System$ is an S-R/R-S cycle, then for all $tr \in Traces(System)$, $Prog(System/tr) \geq 1$.

Base case: $tr = \langle\rangle$: Start state.

All S-R components are ready to send and none of the R-S components are ready, so
$Prog_s(System)$ = the number of S-R components ≥ 1.
There are no messages outstanding, so
$Prog_m(System) = 0$.
$Prog(System)$ = the number of S-R components ≥ 1.

Induction: $tr = tr' \frown \langle e \rangle$.

e is one of send or receive.

If e is receive, then
$Prog_s(System/tr) = Prog_s(System/tr') + 1$ (one more sender active),
$Prog_m(System/tr) = Prog_m(System/tr') - 1$ (one less message pending).
Therefore $Prog(System/tr) = Prog_s(System/tr') + 1 + Prog_m(System/tr') - 1 = Prog(System/tr') \geq 1$.

If e is send, then $Prog_s(System/tr) = Prog_s(System/tr') - 1$ (one less sender active).

There are two cases for $Prog_m(System/tr)$.

If the message connector over which the message is to be sent is not full, then
$Prog_m(System/tr) = Prog_m(System/tr') + 1$.
In this case, $Prog(System/tr) = Prog_s(System/tr') - 1 + Prog_m(System/tr') + 1$
$= Prog(System/tr') \geq 1$.

If the message connector is full, then the message is dropped and
$Prog_m(System/tr) = Prog_m(System/tr')$.
But in this case, we know that $Prog_m(System/tr') \geq 1$, since there must be at least one message pending (to fill the buffer), and
$Prog_s(System/tr) \geq 0$, since there can never be a negative number of ready processes, and therefore
$Prog(System/tr) = Prog_s(System/tr) + Prog_m(System/tr) \geq 1$.

Since progress can be made whenever $Prog(System/tr)$ is non zero, this means that deadlock will not occur. *Q.E.D.*

Notice how this proof proceeds using a standard CSP proof technique: induction over the traces of the process $System$. The structure of the induction step is guided, however by the structure of the architecture, using the property that the total process, $System$, is constructed only of R-S and S-R components and Message connectors.

This proof allows us to define an initial deadlock free style. It defines the interface, component, and connector types, as above, and defines the following constraints ($JC\_Constraints\_2a$):

$$\begin{aligned}
& JC_Constraints_1a \\
\wedge\ & Cycle(Components) \\
\wedge\ & \#Components = \#Connectors \\
\wedge\ & \exists\, c : Components \mid Type(c) = \mathsf{S\text{-}R}
\end{aligned}$$

Any system constructed in this style (*i.e.*, where we can show that the vocabulary and topological constraints hold) is guaranteed to be globally deadlock-free by the theorem $JC\_Theorem\_2a$.

Unfortunately, this analysis, and hence the style derived from it, does not cover many of the systems that we would like to construct. In particular, it only covers systems consisting exactly of a cycle of components. We will have to defer a more general style to the next section.

5.4 Generalizing the JC Style

In the previous section we saw how formalizing a style provides three important benefits to users of the style. First, it clarifies the semantics of the style. By describing both the connector and component types formally, the semantics of systems that use that vocabulary is made precise, and the system descriptions can be checked for consistency as well as analyzed for behavioral properties. Second, formalizing a style clarifies exactly what systems are members of the style. By providing precise constraints on membership in the style, it can be precisely determined whether a given system description obeys those constraints or not. Third, formalizing a style provides a basis for analysis. By considering properties that are shared by all systems in the style, the style designer can reduce the effort of developers of individual styles and provide design guidance to those developers. In the previous section, we showed how it could be proved that all systems in the JC Style are deadlock-free, by virtue of the semantics of the vocabulary used and the constraints on topology imposed by the style. Any developer who uses the style can now be assured that his or her system has this property.

As we also learned from our exercise, formalizing an existing style can reveal limitations of the style. In the case of the JC style, we saw that the analyses performed by Justo and Cunha only considered systems consisting exactly of a cycle of components. Thus, the style, in order to provide guarantees of global deadlock-freedom, must exclude systems that a developer might reasonably like to build, and which are undoubtedly deadlock-free. A system consisting of a sequence of components that does not cycle, for example, is deadlock-free because the first component in the sequence is never prevented from receiving input by anything within the system.

Another limitation of the style as described in the previous section is the rigidity of the R-S and S-R templates. While many components can be forced into those patterns, it might be reasonable to consider alternatives where the pattern of mixing sends and receives is more complicated.

In this section we consider how the R-S and S-R component types can be generalized into a single form. We then show that this new form leads to a guarantee of globally deadlock-free configurations in a more general JC style.

By looking at the proof of global deadlock-freedom in the previous section, we observe that the way in which deadlock occurs (or is avoided) is based on the fact that the R-S and S-R templates control the *dependency* of a component's sending ports on its receiving ports. That is, once a component has completed its initialization (possibly including some preliminary sends), a component can only send on each of its output ports after it has received on all of its input ports. The difference between an R-S and an S-R component is simply whether its initialization includes output messages or is immediately blocked waiting for input. The templates expose these input-output dependencies in a clear way.

Unfortunately, the templates also enforce gratuitous dependencies on the ordering of receives and of sends. That is, the templates specify that they must always send their messages in the same order and receive them in the same order. Logically, there is no good reason to force a component to be structured this way, and there are two important reasons to remove this rigid structure.[2]

First, the details of the ordering constraint do not affect the overall deadlocking behavior of a configuration. The benefits of the style could be extended to many more systems if the ordering limitation were removed. Second, by limiting the potential concurrency of a component, it limits the extent to which a component can be decomposed within the style. A configuration of multiple templates does not fix the order in which its input and output messages will be dealt with, and so maintaining the constrained ordering would not permit the encapsulation of a configuration as a component in another system (or, conversely, the decomposition of a component into a more structured configuration).

The benefits of encapsulation, and its related activity, decomposition, are well known to designers of complex systems, permitting clean top-down design, separation of concerns, and abstraction. If a system designer can cleanly separate parts of the design and perform analysis on aggregates without worrying about the internal structure of each element, this greatly simplifies the tasks of design and analysis. Similarly, it frees the designer of individual components to consider more alternatives without having many constraints imposed on them by the context in which the component will occur in the total system.

If a style obeys the principle of encapsulation, there is another important benefit for the designer of the style. Styles that permit the free encapsulation of configurations are *closed under composition*. This means that any combination of components and connectors in the style is equivalent to some individual component in the same style. For the purposes of analysis, this means that any property that is shown to hold for individual components in the style *also holds for every configuration in the style.*

5.4.1 General Form of Component Type

We now consider a general form of the R-S and S-R components that is closed under composition via the Message connector. This will permit us to establish the constraints necessary to guarantee global deadlock-freedom for a less restrictive form of the JC style.

[2] It is likely that Justo and Cunha chose this structure for implementation reasons. That is, they provided rigid code templates because these are easier to use and test than a more flexible rule.

Component JCM(r:1..;s:1..;F : $((1..r) \times (1..s)) \nrightarrow ((0..) \times (* \mid 1..))$) =
  Port $\mathsf{In}_{1..r}$ = ReceiverType
  Port $\mathsf{Out}_{1..s}$ = SenderType
  Computation = $\|\ (i, j, k, m) : F \bullet \mathsf{C}_k^m(\overline{\mathsf{In}_i.\mathsf{receive}}, \overline{\mathsf{Out}_j.\mathsf{send}})$

Figure 5.7: General Message Style component.

5.4.1.1 New Component Type

Focussing on the issue of dependencies between send and receive ports, we will construct the general component from simple building blocks that represent such a dependency. These simple building blocks will then be combined to represent the full complexity of a component, with dependencies of collections of send ports on collections of receive ports. The R-S and S-R components are simple cases of this, where each send port depends on every receive port. The basic building blocks are defined as follows:

$$\mathsf{C}_i^{max}(\mathsf{r},\mathsf{s}) = \begin{cases} \mathsf{r} \rightarrow \mathsf{C}_1^{max}, & \textbf{when } i = 0 \\ \mathsf{r} \rightarrow \mathsf{C}_{i+1}^{max} \;\Box\; \mathsf{s} \rightarrow \mathsf{C}_{i-1}^{max}, & \textbf{when } 0 < i < max \\ \mathsf{s} \rightarrow \mathsf{C}_{max-1}^{max}, & \textbf{when } i = max \end{cases}$$

$$\mathsf{C}_i^{*}(\mathsf{r},\mathsf{s}) = \begin{cases} \mathsf{r} \rightarrow \mathsf{C}_1^{*}, & \textbf{when } i = 0 \\ \mathsf{r} \rightarrow \mathsf{C}_{i+1}^{*} \;\Box\; \mathsf{s} \rightarrow \mathsf{C}_{i-1}^{*}, & \textbf{otherwise} \end{cases}$$

The process C_i^{max} ensures that there is a receive for each send, possibly excepting a fixed number of initial sends. Initial sends are permitted by using a subscript other than 0 as the initial process. The superscript max indicates that the process can only store that many messages before it must send. C_i^* represents the case where an arbitrary number of messages can be stored.

The generic component (JCM, shown in figure 5.7) is a collection of these in parallel, one for each pair of receive and send ports that have a dependency. The JCM component takes as its parameter a function that maps pairs of receive and send ports (indicated by the set $((1..r) \times (1..s))$) into zero or more sends that are permitted (0..) and either a maximum number of received messages that can be buffered or $*$, indicating that buffering is unlimited (indicated by the set $(* \mid 1..)$). By combining individual C_i^{max} building blocks, more general $m \times n$ dependencies are constructed.

For example, if an output port S depends on two input ports R_1 and R_2, this would be represented using two C_i processes: $\mathsf{C}_i^m(\mathsf{R}_1.\mathsf{receive},\mathsf{S}.\mathsf{send}) \| \mathsf{C}_j^n(\mathsf{R}_2.\mathsf{receive},\mathsf{S}.\mathsf{send})$. The start state (what subscript k is selected for the initial process C_k^{max}) shows whether it is an R-S or an S-R template (see figure 5.7). The generalized version of the R-S component template would use a fixed value of 0 for k, and an S-R component would use a fixed value of 1 for k. In both cases a fixed value of 1 is used for max.

Thus, the following definitions apply in the generalized style:

Component R-S(r:1..; s:1..) = JCM(r,s,$1..r \times 1..s \mapsto (0, 1)$)
Component S-R(r:1..; s:1..) = JCM(r,s,$1..r \times 1..s \mapsto (1, 1)$)

Other simple component types include a trivial component that has no dependencies between any components and one that is similar to R-S in that it must have a receive for every send, but that can buffer its sends arbitrarily.

Component Trivial(r:1..; s:1..) = JCM(r,s,$\emptyset$)
Component Buffering(r:1..; s:1..) = JCM(r,s,$1..r \times 1..s \mapsto (0, *)$)

5.4.1.2 Generalized Constraints

With the new, general, component type we can reconsider what constraints to use to define the new, general, style. Just as $JC\_Constraints\_1a$ defined the constraints that all components had to be R-S or S-R components and that all connectors were Message connectors, we need equivalent constraints using JCM, which we will refer to as $JC\_Constraints\_1b$:

$$\begin{array}{l} \forall\, c : Components \bullet Type(c) = \mathsf{JCM} \\ \wedge\ \forall\, cn : Connectors \bullet Type(cn) = \mathsf{Message} \end{array}$$

We also need to consider what additional constraints are now appropriate, since we want to permit more systems than the restrictive $JC\_Constraints\_2a$ permitted. To propose these rules, we must analyze systems that obey $JC\_Constraints\_1b$ to figure out when they might deadlock. First, let's consider when a single JCM component might deadlock.

We observe that C_i^{max}(receive,send) refuses events in two situations: if $i = 0$, then send will be refused. If $i = max$, then receive will be refused. We must therefore ensure that there will never be a collection of C processes where one set refuses the send events (by having $i = 0$) and the other refuses the receive events (by having $i = max$). Here is one sufficient rule (JCM_Legal):

$$\begin{array}{l} \forall\, c : Components;\ r, s : 1..;\ F : ((1..r) \times (1..s)) \nrightarrow ((0..) \times (* \mid 1..)) \\ \quad \mid Type(c) = \mathsf{JCM}(r, s, F) \\ \quad \bullet \forall\, i : 1..s \bullet \neg\ \exists\, j, k : 1..r;\ m : (* \mid 1..);\ m' : 1.. \\ \qquad \bullet F(j, i) = (0, m) \wedge F(k, i) = (m', m') \end{array}$$

This states that no send event may have both a receive that must precede it ($F(j, i) = (0, m)$) and a receive event that may not occur until after the send ($F(j, k) = (m', m')$). Thus, there can be no cycles of events where each must be preceded by another.

Now that we have established a rule that guarantees that a component is deadlock-free, we can use this to develop a rule for deadlock-free configurations. In order to do so, we will show that, with one significant exception, the JCM component is closed under composition via the Message connector.

What we mean by "closed under composition" is that any configuration of JCM components and Message connectors can be encapsulated as an equivalent JCM component. That is, if we hide all of the attached interfaces (i.e. ports on components that are attached to a Message connector), the process defined by a configuration has the form of a **Computation** for some JCM component.

Figure 5.8: Composition of two components.

5.4.1.3 Proof of Encapsulation

The proof that any configuration can be viewed as a component is by induction on the size of the configuration. There are two steps in the inductive proof. First we consider the case of adding a new component to a configuration, and second we consider adding a connector to a configuration.

For adding a component, refer to figure 5.8. In this case, because there is no connector between the system and the component, the meaning of the new configuration is simply the old system in parallel with the new component:

$$\mathtt{JCM\_Config} \| \mathtt{JCM\_Comp}$$

By the induction assumption, both of these processes have the form $\mathsf{C}_i^m \| \mathsf{C}_j^n \| ...$, so the new configuration is also:

$$\mathsf{C}_i^m \| \mathsf{C}_j^n \| ... \| \mathsf{C}_k^l$$

To show that this is a legal JCM component under the configuration rule JCM_Legal, note that all of the send events in the component and the configuration are distinct, and therefore there can be no conflicting dependencies.

The second case is adding a connector. By the induction hypothesis, the rest of the system is equivalent to a single JCM component, and so adding a connector to a configuration is the same as as adding a connector between two ports on the same component. This is illustrated in figure 5.9 where the two ports to be connected are r and s. The question in this case is whether the behavior at the ports that remain (*e.g.*, ports r' and s' in figure 5.9) have the correct form of parallel dependencies. The essence of the proof is to show that the original dependencies, labelled d_1, d_2, d_3, and d_4, together with the new connector M result in a proper dependency between the ports r' and s' in the new system. This is true except when d_1 is $\mathsf{C}_0^m(r, s)$.

The reason for this is that the pathway $d_3 \rightarrow \mathsf{M} \rightarrow d_2$ defines a single dependency of the correct form. Thus, the total dependency is simply the more restrictive of that pathway and d_4. The only problem that would arise is when d_1 prevents the pathway from being initialized. In effect, if d_1 is too restrictive, two dependencies create a problem analogous to a violation of JCM_Legal.

There are two cases to consider. First, if there is no dependency between r and s, and second, if there is a dependency $\mathsf{C}_k^m(r, s)$, where $k > 0$.

If $M_a(s, r)$ is a message buffer connector from s to r containing a messages, we show that

$$
\begin{array}{rl}
 & (\mathrm{C}_i^m(\mathrm{r}',\mathrm{s}) \| \ \mathrm{C}_j^n(\mathrm{r},\mathrm{s}') \ \| \ \mathrm{C}_k^l(\mathrm{r},\mathrm{s}) \ \| \ \mathrm{M}_0(\mathrm{s},\mathrm{r})) \setminus \{r, s\} \\
= & \mathrm{C}_{i+j}^{m+n+k}(\mathrm{r}',\mathrm{s}') \\
and & \\
 & (\mathrm{C}_i^m(\mathrm{r}',\mathrm{s}) \| \ \mathrm{C}_j^n(\mathrm{r},\mathrm{s}') \ \| \ \mathrm{M}_0(\mathrm{s},\mathrm{r})) \setminus \{r, s\} \\
= & \mathrm{C}_{i+j}^{*}(\mathrm{r}',\mathrm{s}')
\end{array}
$$

where $* + x = *$ for any x.

To prove this, we will show that the failures of each process are identical.

First, we need a useful lemma:

Lemma 1 *After any trace of $C_k^m(r,s) \| M_0(s,r)$, in the new state $C_{k'}^m(r,s) \| M_a(s,r)$, $k' + a = k$.*

Proof of lemma: Observe the state change for each of the events s and r. If we begin in a state $\mathrm{C}_i^m \ \| \mathrm{M}_a$ and proceed via s to $\mathrm{C}_{i'}^m \ \| \mathrm{M}_{a'}$, then $i' = i - 1$ (one less extra message in the component) and $a' = a + 1$ (one more message in the connector). Thus $i' + a' = i + a = k$. If we proceed via an r event, then $i' = i + 1$ (one more message in the component) and $a' = a - 1$ (one less in the connector). Thus again, $i' + a' = i + a = k$. *Q.E.D.*

We now consider the failures of the full composition, by induction on the length of a trace. Let t be a trace, $(\mathrm{C}_i^m(\mathrm{r}',\mathrm{s}) \| \mathrm{C}_j^n(\mathrm{r},\mathrm{s}') \| \mathrm{C}_k^l(\mathrm{r},\mathrm{s}) \| \mathrm{M}_0(\mathrm{s},\mathrm{r})) / t = \mathrm{C}_b^m(\mathrm{r}',\mathrm{s}) \| \mathrm{C}_d^n(\mathrm{r},\mathrm{s}') \| \mathrm{C}_c^n(\mathrm{r},\mathrm{s}) \| \mathrm{M}_a(\mathrm{s},\mathrm{r})$, and $\mathrm{C}_{i+j}^{m+n+k}(\mathrm{r}',\mathrm{s}') \ / (t \ \restriction \ \{r', s'\}) = \mathrm{C}_u^{m+n+k}(\mathrm{r}',\mathrm{s}')$. We will show that for any $t \in \{r, s, s', r'\}*$, $u = b + d + a$ and that the refusals of the two processes are identical.

Base Case: $t = \langle\rangle$.

For trace $t = \langle\rangle$, $b = i \wedge d = j \wedge a = 0 \wedge u = i + j \Rightarrow u = b + d + a$.

Induction: $t = t' \frown \langle e \rangle$.

Let $(\mathrm{C}_i^m(\mathrm{r}',\mathrm{s}) \ \| \mathrm{C}_j^n(\mathrm{r},\mathrm{s}') \| \mathrm{C}_k^l(\mathrm{r},\mathrm{s}) \ \| \mathrm{M}_0(\mathrm{s},\mathrm{r})) \ / t' = \mathrm{C}_{b'}^m(\mathrm{r}',\mathrm{s}) \| \mathrm{C}_{d'}^n(\mathrm{r},\mathrm{s}') \| \mathrm{C}_{c'}^n(\mathrm{r},\mathrm{s}) \| \mathrm{M}_{a'}(\mathrm{s},\mathrm{r})$, and $\mathrm{C}_{i+j}^{m+n+k}(\mathrm{r}',\mathrm{s}') \ / (t' \restriction \{r', s'\}) = \mathrm{C}_{u'}^{m+n+k}(\mathrm{r}',\mathrm{s}')$. By induction hypothesis, $b' + d' = u'$.
If $e = \mathrm{r}'$ then $b = b' + 1$, $d = d'$, $a = a'$, and $u = u' + 1 = (b' + 1) + d' + a' = b + d + a$.
If $e = \mathrm{s}'$ then $b = b'$, $d = d' - 1$, $a = a'$, and $u = u' - 1 = b' + (d' - 1) + a' = b + d + a$.
If $e = \mathrm{r}$ then $b = b'$, $d = d' + 1$, $a = a' - 1$, and $u = u' = b' + d' + 1 + a' - 1 = b + d + a$.
If $e = \mathrm{s}$ then $b = b' - 1$, $d = d'$, $a = a' + 1$, and $u = u' = b' - 1 + d' + a' + 1 = b + d + u$.

So $b + d + a = u$ after any trace t.

We must now show that the refusals are the same after any trace, and thus that the set of traces are the same. Suppose that after trace t we have the processes $\mathrm{C}_b^m(\mathrm{r}',\mathrm{s}) \| \mathrm{C}_d^n(\mathrm{r},\mathrm{s}') \| \mathrm{C}_c^n(\mathrm{r},\mathrm{s}) \| \mathrm{M}_a(\mathrm{s},\mathrm{r})$ and $\mathrm{C}_u^{m+n+k}(\mathrm{r}',\mathrm{s}')$ respectively. $\mathrm{C}_u^{m+n+k}(\mathrm{r}',\mathrm{s}')$ refuses s′ exactly when $u = 0$, and r′ exactly when $u = m + n + k$.

If $u = 0$, by our previous argument, $b + d + a = 0$. This means that $b = 0 \wedge d = 0 \wedge a = 0$. Because $d = 0$, s′ may not be accepted immediately. This means that there must be an r before it. But since $a = 0$, r must be preceded by an s. But since $b = 0$, s must be preceded by r′, and so s′ is refused until r′ occurs.

If $u > 0$, then one of b, d or $a > 0$. Suppose $d > 0$. Then s′ is not refused.
Suppose $a > 0$. Then M_a does not refuse r. Also, by our lemma, $c = k - a < l$, and so $\mathrm{C}_c^l(\mathrm{r},\mathrm{s})$ does not refuse r. Thus, if $\mathrm{C}_d^n(\mathrm{r},\mathrm{s}')$ does not refuse r, it can occur, after which

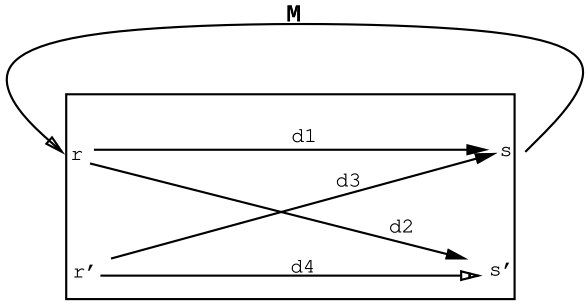

Figure 5.9: Composition of a component and a connector.

$d' = d + 1 > 0$, and therefore s′ is not refused.
If C_d^n(r,s′) does refuse r, then $d = n > 0$, and therefore s′ is not refused. If $a = 0$, $d = 0$, and $b > 0$, then $c = k > 0$, so s is possible, after which $a' = a + 1$, which case has already been discussed. Thus, the refusals of the event s′ are identical.

If $u = m + n + k$, then $b = m$, $d = n$, and $a = k$. $b = m$ implies that no immediate r′ is possible, an s must precede it. but $a = k \Rightarrow c = 0$ which implies no immediate s is possible, an r must precede it. But $d = n$ implies that no immediate r is possible, but an s′ must precede that, and therefore r′ is refused, just as for C_u^{m+n+k}(r′,s′)

If $u < m + n + k$, then $b < m$ or $d < n$ or $a < k$.
If $b < m$, then an immediate r′ is possible.
If $b = m \wedge a < k$, then $c = k - a > 0$ and an s event is possible, after which $b' = m - 1 < m$.
If $b = m \wedge a = k \wedge d < n$ then an r event is possible, after which $b' = m \wedge a' = a - 1 < k$.

Thus the refusals of the event r′ are identical, which concludes our proof that

$$\begin{aligned} & C_i^m(r',s) \| C_j^n(r,s') \| C_k^l(r,s) \| M_0(s,r) \setminus \{r, s\} \\ = \; & C_{i+j}^{m+n+k}(r',s'). \end{aligned}$$

The proof that

$$\begin{aligned} & C_i^m(r',s) \| C_j^n(r,s') \| M_0(s,r) \setminus \{r, s\} \\ = \; & C_{i+j}^{*}(r',s') \end{aligned}$$

proceeds similarly with the difference that there is no process C_c^l(r,s) to refuse s events. Thus, r′ is never refused, because whenever $b = m$, an s event can occur so that $b' = m - 1 < m$.

It now remains to show that the resulting parallel composition of C_i^m processes is a legal JCM component. There are two aspects of this: First, we must show that there is at most a single dependency between each receive-send port pair, and second, we must show that the configuration obeys JCM_Legal.

To handle the case where there is already a dependency between r′ and s′, we need simply observe that for any i, j, m, n:

C_i^m(r,s) $\|C_j^n$(r,s) = C_k^l(r,s) where $k = min(i, j)$ and $l = min(m - i, n - j) + k$.

The resulting process is defined whenever $l > 0$. This is always the case because we know (by hypothesis) that for the constructed process C_i^m, $m > i$, and thus $m - i > 0$. Thus, the encapsulating component can be represented using the JCM template.

To show JCM_Legal, we must consider whether, when the constructed dependency is C_0^m(r′,s′), there can be another dependency C_n^n(r″,s′) or, when the dependency is C_m^m(r′,s′) there can be a dependency C_0^n(r″,s′).

The first situation, C_0^m(r′,s′), can only arise if either the dependency d_4 or the dependency d_2 was C_0. In this case, by JCM_Legal for the original component, there can be no other dependency C_n^n(r″,s′), and so JCM_Legal holds for the encapsulation. The second situation, C_m^m(r′,s′), arises only when the dependency d_4 is of this form. Again, this precludes any other dependencies C_0^n(r″,s′) in the original and thus the encapsulated component.

This concludes the proof of the following theorem about JC style configurations:

Any configuration of JCM components and Message connectors not containing a cycle of C_0 processes is equivalent to a single JCM component.

To state this formally, we define a cycle of C_0 processes:[3]

$$
\begin{aligned}
&\forall\, CN : \mathrm{seq}\, Connectors \bullet C_0\_Cycle(CN) \mathrel{\widehat=} \\
&\qquad \forall\, d : 1..\#S - 1 \bullet \exists\, c : Components;\ i, j, k, l : 1..;\ n : (* \mid 1..) \mid \\
&\qquad\quad ((c, \mathsf{In}_i), (S(d), \mathsf{Receiver}_j)) \in attachments \\
&\qquad\quad \wedge\ ((c, \mathsf{Out}_k), (S(d+1), \mathsf{Sender}_l)) \in attachments \\
&\qquad\quad \wedge\ (i, k, 0, n) \in F(c) \\
&\qquad \wedge\ \exists\, c : Components, i, j, k, l : 1..;\ n : (* \mid 1..) \mid \\
&\qquad\quad ((c, \mathsf{In}_i), (S(\#S), \mathsf{Receiver}_j)) \in attachments \\
&\qquad\quad \wedge\ ((c, \mathsf{Out}_k), (S(0), \mathsf{Sender}_l)) \in attachments \\
&\qquad\quad \wedge\ (i, k, 0, n) \in F(c)
\end{aligned}
$$

where $F(c)$ is the dependency function of JCM component c

Using this definition, we can state the theorem formally:

$$
\begin{aligned}
&\ (\forall\, c : Components \bullet Type(c) = \mathsf{JCM} \wedge JCM\_legal(c) \\
&\wedge \forall\, cn : Connectors \bullet Type(cn) = \mathsf{Message} \\
&\wedge \neg\ \exists\, S : \mathrm{seq}\, Connectors \mid C_0\_Cycle(S)) \\
&\Rightarrow \exists\, r, s : 1..;\ F : \mathbb{P}(1..r \times 1..s) \times (0.. \times (* \mid 1..));\ f : \alpha System \rightarrowtail \alpha\mathsf{JCM}(r, s, F) \\
&\qquad \bullet f(System) = \mathsf{JCM}(r, s, F) \wedge JCM\_legal(f(System))
\end{aligned}
$$

5.4.1.4 New Style Definition

This result justifies a new recasting of the constraint $JC\_Constraints\_2a$ in the more general style ($JC\_Constraints\_2b$):

$$\neg\ \exists\, S : \mathrm{seq}\, Connectors \mid C_0\_Cycle(S)$$

The full constraints for a general, JCM based style (JCM_Style), now consist of:

[3] We can't reuse *Cycle* here because we need to identify the dependencies on the specific ports used to connect the cycle. Thus, we identify the connectors used and thus the ports.

$$\begin{array}{l} \quad \forall\, c : Components \bullet Type(c) = \mathsf{JCM} \\ \wedge\ \forall\, cn : Connectors \bullet Type(cn) = \mathsf{Message} \wedge \forall\, c : Components \bullet JCM\_Legal \\ \wedge\ \neg\ \exists\, S : \mathrm{seq}\ Connectors \mid C_0\_Cycle(S) \end{array}$$

and the following theorem holds about the style:

$$JCM\_Style \Rightarrow Deadlock\_Free$$

If we wish to consider only systems consisting of R-S and S-R components (for implementation reasons, perhaps), we can define a style JC_Style with this set of constraints:

$$\begin{array}{l} \quad JC\_Constraints\_1\,a \\ \wedge\ \forall\, C : \mathbb{P}\ Components \mid Cycle(C) \bullet \exists\, c : C \bullet Type(c) \neq \mathsf{R\text{-}S} \end{array}$$

This is a *substyle* of JCM_Style, meaning that $JC\_Style \Rightarrow JCM\_Style$. Thus, we know that

$$JC\_Style \Rightarrow Deadlock\_Free.$$

These results justify and extend the configuration rules proposed by Justo and Cunha in their informal treatment of the style. While including the special cases that they proved informally, of systems consisting only of a single cycle, it also covers the more complex cases that violated their simple rules. This generalized model also has the advantage of extending to cover other templates besides the simple S-R and R-S cases. It would handle, for example, a component in which only *some* of the output ports were initialized by sending before receives were necessary, or cases where only some of the inputs were needed before sending could begin.

In their paper, Justo and Cunha discuss ways to use their rules and templates to avoid deadlock, and in their example they are forced to introduce extra *dummy* messages to convert one of the R-S components into an S-R template so that they break a cycle. In the more general system, deadlock could be avoided by recognizing that the component in question only has to wait for a message that is not relevant to the cycle, and so there is no need either for deadlock or for dummy messages.

5.5 Discussion

In this chapter we have shown, through the JC case study, how using WRIGHT to formalize a family of systems as an architectural style can provide important benefits for software architects. First, the formalization clarifies the semantics of the style. The meaning of specialized vocabulary is made precise as component and connector types. Second, formalizing a style clarifies what configurations are members of the style. It can be precisely determined whether a configuration obeys the constraints that define the style. Third, formalizing the style provides a basis for analysis. We have shown how it can be proven that a property holds for every member of a style.

We further showed how, once a style has been formalized, this can reveal weaknesses in the previously informal style, either because it does not after all guarantee the hoped-for properties or because it does not include systems of interest. By discovering these weaknesses, we showed how a style can be tightened to guarantee critical properties and extended to make those guarantees for more systems.

In the course of the case study, we introduced a number of techniques that are generally applicable to the formalization of architectural styles:

- **Separation of Vocabulary and Topological Constraints:** A common form of constraint is one that restricts the vocabulary of types that can be used in configurations of a given style. A particular vocabulary is often the starting point for the development of a style, and it is worth calling out the restrictions on the vocabulary right at the beginning. This can guide users of the style in construction of systems as well as serving as a starting point for analysis. Then, once more is known about systems constructed using the given vocabulary, topological constraints can be added depending on what is discovered.

- **Separation of Style Constraints from Style Properties:** For any style to have value to architects, it must define two different things. First, the constraints that define the style must be defined, so that an architect can determine whether a configuration is a member of the style. Second, the properties that the style developer has proved about the style need to be explicitly stated, so that the architect can use them as assumptions about a configuration. An example of a style *constraint* in the JC style is “no cycles of R-S components.” These are predicates that an architect must prove about a configuration. An example of a style *property* is “the system is deadlock-free.” These are predicates that a configuration developer may assume about a configuration. It is easy to see that the value of a style to architects depends on having weak constraints (that are easy to prove) and strong properties (that provide a lot of information).

- **Trace and Structural Induction:** In proving global deadlock-freedom for various forms of the JC style, we relied on two powerful kinds of induction: trace induction, a general technique from CSP, and structural induction, an architecture-specific technique. In trace induction, we consider all possible states of a CSP process by extending an event trace by a single event. In structural induction, we consider all possible configurations in a style by building up larger configurations using small, discrete steps. Each of these can be applied in a variety of settings and aid the architect in establishing style properties.

- **Encapsulation:** An especially powerful form of structural induction is a proof of *encapsulation*, when any configuration in the style can be shown to be equivalent to a single component in the style. Once encapsulation is established, *any* proof regarding individual components applies immediately to all configurations in the style.

- **Formalize, then Generalize:** We followed a particular method in our formalization of the JC style: First, we formalized an existing, informal style, attempting to be faithful to the informal description. Next, we considered how the power of that style could be extended by generalizing the vocabulary and weakening the constraints. This led us to be able to demonstrate the same, strong result (global deadlock-freedom) over a much wider range of systems. We also used this formalize-then-generalize approach in our treatment of the Aegis system in the previous chapter. There, we formalized the “naive” client-server interaction model and applied it to the initial architectural design. Then, we used that formalization to expose potential problems and to suggest general solutions to the issues raised. In each case, this approach lets us get started on the formalization relatively easily, by using the informal description as a template for the formalization, and then deepen our insight based on analysis of the more tractable, precise formalization.

Chapter 6

Case Study: HLA

6.1 Introduction

In the previous two chapters we have seen how WRIGHT can be used to describe and analyze both architectural configurations and architectural styles. By formalizing a configuration, we expose critical issues about the system and enable the architect both to locate potential problems and to consider design alternatives. By formalizing a style, we support the effective application of architectural design techniques to multiple systems. We can understand the membership of a defined family of systems and provide analyses that apply to all members of the family.

Our configuration case study, AEGIS, also showed how WRIGHT could be applied to complex, real-world systems. We also need to consider how WRIGHT's facilities for the description and analysis of style support analysis of problems that are of an industrial scale. To do so, we will demonstrate, in the context of an actual development effort, the use of a number of techniques to show how the formalization and analysis effort can remain tractable even as the system complexity increases.

These techniques include:

- **Architectural structure**: By mimicking the structure of the software system, WRIGHT partitions the description and analysis of the system into manageable, separable elements.

- **Architectural style**: By focussing on *families of systems* rather than on individual systems, the benefits of the analysis are leveraged across many systems. This increases the total benefit of the effort by allowing many development efforts to take advantage of each result. The total cost of the formalization is amortized over the entire family of systems, reducing the effective cost to any individual system development project.

- **Abstraction**: While a typical software system contains a huge amount of detail, it is often possible to ignore some parts of that detail while focussing on others. This abstraction of the behavior of the system can lead to marked increases in both the comprehensibility of the specification and in the ability to analyze the system.

- **Automated analysis**: By ensuring that an architectural abstraction is *finite state*, it is possible to apply automatic commercial tools such as symbolic model checkers to carry out analysis for us. Thus, the level of effort (and inspiration) required by the human developer is vastly reduced. The automatic tools locate problems in the system, and the developers' level of confidence in the results is vastly increased.

- **Incrementality**: Many formal methods require complete specification of the entire system before useful analysis can take place. Thus, as the complexity of a software system increases, so does the cost of achieving *any* benefit from the formal effort. We will show that it is possible to develop an architectural specification in stages, each stage modelling a new facet of the overall system. At each step, the overall model is augmented with new details about architectural behavior that can be checked in combination with what has already been specified. Since incremental efforts yield incremental payoffs, the benefits of formalization become commensurate with the costs of formalization.
- **Traceability**: In order for architectural analysis to have value, it must be possible to relate the properties discovered in the abstract specification to the system under consideration. We will show how to structure WRIGHT behavior specifications to maintain a correspondence with the structure of the informal document from which they are constructed. In this way, issues uncovered by analysis lead directly to insight into the original, informal specification.

This chapter demonstrates the use of these techniques through a case study, the "High Level Architecture for Simulations (HLA)," [DMSO95] developed through the Defense Modeling and Simulation Office (DMSO). We undertook the formalization described in this case study as part of the early development of this standard, which has recently been released as an official standard [DMSO]. This work is continuing as part of DMSO's efforts to improve the standard, to define acceptance test criteria for implementations, and to develop supplemental documentation of the interface.

In the next section we provide an informal overview of the HLA. Then we will show how the above techniques can be applied to use WRIGHT to analyze a large-scale specification such as HLA. This presentation will begin with an overview of the WRIGHT specification of HLA in section 6.3. Then, the techniques will be illustrated in section 6.4 through a series of example analyses of parts of the HLA. (A "full" WRIGHT specification of HLA is given in appendix B.) In section 6.5 we discuss how the techniques we have seen here can be applied in general to support the analysis of other large software systems.

6.2 Motivation and Overview of HLA

Simulation is an important tool for the military, both in preparing for and in carrying out its various missions throughout the world. Simulations are used in personnel training, design and testing of equipment, and analysis of both past and future actions. The U.S. DoD has made a considerable investment in equipment and software for simulation – one effort alone, directed at army training, cost \$2 billion [Bro96].

Given the number and complexity of these simulations, developers are faced with a daunting task: to provide software that meets the challenges posed by the high-fidelity, real-time, physically distributed, mission-critical simulation domain, and yet to minimize

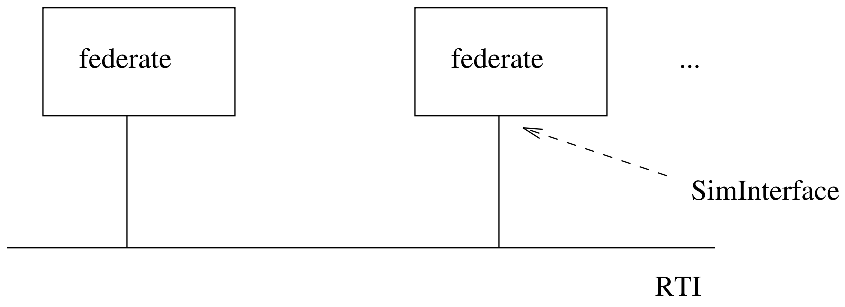

Figure 6.1: A federation.

redundancy of effort across applications and maximize flexibility of the software to be used for new, possibly unanticipated tasks.

With the potential benefits of a well-defined common architecture in mind, the Defense Modelling and Simulation Office (DMSO) has undertaken the development of the "High Level Architecture for Simulations" (HLA) [DMSO]. The HLA is intended to support the coordination of different simulations. The goal is to simplify the integration task, encourage modularization of simulations, and increase quality of simulations and potential for reuse.

The HLA defines a standard for the coordination of individual simulations through the communication of data object attributes and events. In HLA, members of a *federation* — the HLA term for a distributed simulation — coordinate their models of parts of the world through sharing objects of interest and the attributes that define them. Each member of the federation (termed a *federate*) is responsible for calculating some part of the larger simulation and broadcasting updates using the facilities of the Runtime Infrastructure (RTI). Messages both from the federates, *e.g.*, to indicate new data values, and to the federates, *e.g.*, to request updates for a particular attribute, are defined in the "Interface Specification" document [DMSO95]. Each message is defined by a name, a set of parameters, a possible return value, pre- and post-conditions, and a set of exceptions that may occur during execution of the message.

The interface is divided into five parts: Federation management, declaration management, object management, ownership management, and time management.

- *Federation* messages are used by federates to initiate a federation execution, to join or leave an execution in progress, to pause and resume, and to handle saves of execution state.
- *Declaration* messages are used to communicate about what kinds of object attributes are available and of interest.
- *Object* messages communicate actual object values.
- *Ownership* messages are used in situations when one federate has been responsible for calculating the value of an object attribute but for some reason another federate should now take over that responsibility. (Example situations include when the original federate must drop out of a simulation or when some property of the object indicates that the new federate is better able to support that object. For example, if

a unit moves from one geographic region to another, then simulators responsible for modelling troops in each region might hand off ownership of the unit's representation object.)

- *Time* messages are used to keep each member of the federation synchronized, either by maintaining correspondence of wall-clock time, by lock-step advancement of a logical time, or by other means.

The intent of the interface specification is that the general standard be refined into multiple implementations depending on the various needs of particular simulation domains. For example, different simulations would have different performance constraints, requirements for physical distribution, and models of time-synchronization, depending on the scale and use of the simulation. In addition, each federation needs to augment the standard with its own detailed object-model to ensure semantically consistent exchange of data between federates.

For example, as part of the current standard development effort, several implementation efforts, each termed a *proto-federation*, are underway. One proto-federation effort is described in [GG95].

6.3 Overview of the WRIGHT Specification of the HLA

As we described above, an HLA federation consists of a number of federates communicating via the RTI. This view of a federation is shown in boxes-and-lines form in figure 6.1.

In order to apply our architectural abstractions to the description of this system, we observe two things. First, we observe that we are attempting to describe a *family* of systems; that is, the intent of the interface specification (IFSpec) is to provide a standard that applies to many different federations, not just a single federation. Thus, we will describe HLA as an architectural style, rather than as a configuration.

Second, we observe that there are two architectural elements that are captured in the IFSpec and in the diagram of figure 6.1. These are, first, the interface that must be satisfied by any federate in order to participate in interaction via an RTI, and second, the way in which different messages to and from different federates are combined to form a single, coherent simulation via the RTI.

These two elements can be formalized using WRIGHT. The first, the interface, is formalized as an **Interface Type**, which we name SimInterface. The second element, the way that messages are combined to form an interaction, is exactly a connector **Glue**. Thus, we will formalize this as the RTI connector in our style. The RTI formalization will use the SimInterface to describe the roles of the connector, but because of the way that we can use the architectural structure of WRIGHT, the descriptions of the federates' requirements and how federates are combined remain separate. The overall WRIGHT specification of the HLA style (without details) is shown in figure 6.2. In this specification, the fact that we do not constrain the type or number of components in a configuration indicates that components may be varied freely in the style, as long as they communicate solely via the RTI connector.

By separating the SimInterface and RTI descriptions we gain important advantages. First, we can consider them separately. If the IFSpec is to be useful to developers of a

Style HLA
  Interface Type SimInterface = ...
  Connector RTI(*nsims*: 1..)
    Role $\mathsf{Fed}_{1..nsims}$ = SimInterface
    Glue = ...
Constraints
  $\exists\, r : Connectors \mid \{r\} = Connectors \wedge Type(r) = \mathsf{RTI}$
End Style.

Figure 6.2: The HLA Style.

single federate, then it is important to have it made explicit exactly what that federate is required to do and when, and what the federate can expect from the federation as a whole. This is important because one of the goals of the IFSpec is to provide a better means of acquiring simulations from multiple vendors. The idea is that each vendor can build just a single federate that will then be combined into a larger federation. The WRIGHT separation supports this by separating the description of the interaction *from the point of view of a single federate* into a separate **Interface Type** description.

Separating SimInterface and RTI also simplifies the description of the combined behavior. This description doesn't have to re-describe all of the preconditions of each message—that is handled by the SimInterface. Instead, it can focus its description on how messages are related to each other. For example, our description of how an object attribute value is communicated from one component to another is quite simple; it simply specifies that an update message from the owning component is followed by a notification message to each subscribing component. An unstructured description would have to complicate this with constraints such as that this must only occur when federates are joined and when there is no pause in effect. We are thus free to vary these conditions or to perform analysis on the system without these additional complications getting in the way.

6.3.1 Overview of SimInterface

Structuring our description of the HLA as an architectural style, with two main architectural elements, has given us an initial way of attacking the problem of formalizing a large system. Now we must consider how a WRIGHT formalization of each of the two main elements of the description, the SimInterface and the RTI can be approached to provide practical benefits. Our description of the SimInterface is summarized in figure 6.3.

The overall behavior description is divided into six parts. The first four parts correspond to four of the five management groups, while the last two represent relationships among events in different management groups.[1]

We have structured the description as a collection of processes in parallel for two reasons: First, in order to separate different concerns in the specification. Because there are so many different tasks that are undertaken by a federate, we need to break these down into

[1] As we will see below, our specification is a partial specification of the HLA, and deliberately ignores the time management group. Thus, there is no process TimeMgmt in our specification.

$$
\begin{array}{l}
\textbf{Interface Type } \mathsf{SimInterface} = \mathsf{FedMgmt} \parallel \mathsf{DeclMgmt} \parallel \mathsf{ObjMgmt} \parallel \mathsf{OwnMgmt} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{FedJoined} \parallel \mathsf{ControlPause} \\
\quad \textbf{where} \\
\quad \mathsf{FedJoined} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{RUN}_{FedEvents} \,; \\
\qquad\qquad\qquad \overline{\mathsf{resignFedExecution}} \rightarrow \S \\
\quad \mathsf{ControlPause} = \mathsf{RUN}_{NotPausedEvents} \,; \overline{\mathsf{pauseAchieved}} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \mathsf{ControlPause} \\
\quad \mathsf{RUN}_S = \S \;\square\; (\square\, e : S \bullet e \rightarrow \mathsf{RUN}_S)
\end{array}
$$

Figure 6.3: WRIGHT description of SimInterface.

different constraints that can be handled separately. Otherwise, it would be too difficult to understand and analyze the specification.

For example, consider the process FedJoined shown in figure 6.3. This process specifies that a federate may only perform certain operations when it is *joined* to the federation execution.

A sub-process controls certain aspects of how an event can occur by including it in the process' alphabet. Recall that the alphabet of a process is the set of events that are mentioned in the process' specification. A sub-process controls an event in the overall specification because every process that mentions that event must *synchronize* before the given event can occur. So for example, FedJoined indicates that the first event it permits is joinFedExecution. That means that no other event in FedJoined's alphabet may occur before joinFedExecution. The alphabet of FedJoined is joinFedExecution, resignFedExecution, and all of the events in the set $FedEvents$.[2] Thus, no event in $FedEvents$ can occur before the federate joins or after it resigns, and certainly the federate may not resign if it is not joined.

Once the federate has joined the federation execution, FedJoined acts as $\mathsf{RUN}_{FedEvents}$. This means that any event in $FedEvents$ is free to occur (according to this specification). Because RUN is deterministic, FedJoined does not constrain which of these events will occur. That is left to other parts of the specification. FedMgmt will control when pauses and resumes occur, ObjMgmt will control when updates occur, *etc*.

FedJoined also specifies that resignFedExecution is the last event (in FedJoined's alphabet) that will occur. It does this by placing it after (via ';') RUN and becoming $\S$ after it occurs. Because RUN terminates with an *external* choice, FedJoined specifies that some other part of the specification controls when resignFedExecution will occur; FedJoined only specifies what *may not* happen after it occurs.

Because we have separated this constraint into a single sub-process, none of the other processes need to worry about this constraint. If we had not done this, we would have had to include the joinFedExecution and resignFedExecution events in each of the processes for the management groups. This way, they can be separated. FedJoined ensures that joinFedExecution will occur before any of the $FedEvents$, without the intervention of every other part of the specification.

The second reason to structure the specification via interacting sub-processes is to

[2]The definition of *FedEvents* and all other definitions omitted in this chapter are found in appendix B.

provide *traceability* from the WRIGHT specification to the IFSpec document from which we are working. We have chosen to keep the structure of the WRIGHT specification the same as that of the IFSpec. The IFSpec is divided into management groups; our WRIGHT specification is divided into management groups. The IFSpec considers the constraints between different messages in separate sections; the WRIGHT specification breaks out the constraints between messages into separate sub-processes. What this means is that if we discover a behavior in the WRIGHT specification, we can see how the different sub-processes contribute to that behavior, and thus how those behaviors arise because of descriptions found in the original IFSpec; each sub-process relates back to a particular part of the IFSpec document.

By placing each of these sub-processes together in a single, composed specification, on the other hand, we can discover interactions that result from the *combination* of properties that appear at widely separated points in the IFSpec. In an informal specification, this kind of interaction is very difficult to locate and understand, because only one part of the specification is visible at a time, and different terms may be applied to overlapping concepts. By using a formal specification where we can apply automated analysis tools, we are able to discover these emergent properties.

Another aspect that is visible from the structuring of SimInterface is how we have *abstracted* from the full HLA specification. Note that we have provided only *four* of the *five* management groups in our specification. This is an important property of the specification for a number of reasons. First, the fifth group, time management, is probably the most complex group of the IFSpec. If we had to include all of the elements of time management in our specification before we could get any results, we would have significantly increased the cost of performing the formalization. Second, the specification is not invalidated by the absence of these messages. Any issues or behaviors that we discover from the current specification are valid in the full specification; we just can't discover properties that relate directly to the presence of the time management messages. Third, a full specification of the HLA that included time management would include the current specification as a subset. We could add the time related messages as an add-on without redoing the rest of the specification.

Thus, our WRIGHT specification of the HLA permits an *incremental* approach to formalizing the system. We can specify some of the HLA and get immediate, valid results, without incurring the cost of a full specification. Then, if we choose, we can add more to the specification and get more results without having to redo what we've already accomplished. Again, we get more results for a proportionate additional effort.

6.3.2 Overview of RTI Glue

In this section we provide an overview of the RTI **Glue**. The glue is shown in figure 6.4.

Like SimInterface, our description of the RTI glue uses sub-processes to separate different elements of the glue's behavior description. As for the SimInterface the sub-processes can be divided into local concerns (represented by HandleSims) and global constraints (represented by FedExists and WhatSims).

In this case however, because the **Glue** is concerned with how federates are combined rather than how a single federate operates, the "local" concerns are different. Rather than

$$
\begin{array}{l}
\textbf{Glue} = \mathsf{FedExists} \parallel \mathsf{WhatSims}_S \parallel \mathsf{HandleSims} \\
\quad \textbf{where} \\
\qquad \mathsf{WhatSims}_S = \mathsf{checkSims!S} \rightarrow \mathsf{WhatSims}_S \\
\qquad\qquad\qquad \Box\, (\Box\, i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \mathsf{WhatSims}_{S\cup\{i\}}) \\
\qquad\qquad\qquad \Box\, (\Box\, i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{WhatSims}_{S-\{i\}}) \\
\\
\qquad \mathsf{HandleSims} = \mathsf{HandlePause} \parallel \mathsf{HandleResume} \parallel \mathsf{SaveFed} \parallel \mathsf{RestoreFed} \\
\qquad\qquad\qquad \parallel \mathsf{NewObject} \parallel \mathsf{OwnDivestiture} \ldots
\end{array}
$$

Figure 6.4: The RTI Glue.

representing the structure of the interface, each element of HandleSims represents a separable "mini-protocol" that shows how a single event initiated by a federate is communicated to other federates, and how those other federates must respond. That is, each mini-protocol shows the relationship between the different components and how their individual actions are combined to create a single, coherent simulation. For example, consider the process NewObject:

$$
\begin{array}{l}
\mathsf{NewObject} = \Box\, i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{instantiateObj} \rightarrow \mathsf{checkSims?S} \\
\qquad\qquad\qquad\qquad\qquad\quad \rightarrow (\sqcap\, J : \mathbb{P}\, S \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad (;\, j : J \bullet \overline{\mathsf{Fed}_j.\mathsf{instantiateDiscoveredObj}} \rightarrow \S)); \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{NewObject}
\end{array}
$$

This mini-protocol specifies what happens when a federate creates a new object. Whenever a federate sends the message instantiateObj, some set of the currently joined federates will be informed of the new object by the message instantiateDiscoveredObj.

Notice again how we have carefully *abstracted* from the full details of the IFSpec. The specification of NewObject indicates (via the internal choice of the set J) that *some* subset of the joined federates will be notified of the new object's existence, in some order. In fact, the IFSpec indicates that there is a particular set of federates that must be notified: Those that subscribe to that class of object and whose "discovery predicates" are satisfied by the new object.

This is another example of how we have simplified the specification by focussing on a particular aspect while ignoring others. In this case, we have chosen to represent the relationship between events (that instantiateObj causes the sending of zero or more instantiateDiscoveredObjs) while ignoring the complex data-state that must be maintained by an actual RTI implementation. The size of the state of declarations and discovery predicates would be quite large and could make it impossible for automated analysis tools to be applied to this specification. This is because there must be an element of state for each federate for each object class (whether the federate subscribes to that class). This is a potentially unbounded amount of information, and thus we must abstract away from this complexity in order to provide tractable analysis.

Further, this kind of structured data-intensive property is more easily specified in formalisms other than CSP. It makes sense to carve off that part of the problem, indicating how different data structures are related and ensuring that state invariants are maintained consistently, for specification and analysis in a different view of the architecture, based on a different underlying semantics such as Z [Spi92]. We will show an example of how this might be done in chapter 8.

Using non-determinism to elide details provides a nice balance between complexity and fidelity. It reduces complexity because no state must be maintained to indicate which subset is chosen, but it indicates that a choice is made. The correct choice will be a refinement of this abstract specification, and could be specified and analyzed in a more detailed specification. As with the omission of the time management messages, any results of analysis on this simpler specification will still hold true of the more faithful, more complex specification.

Each of the mini-protocols is similar to NewObject. First, some event is initiated by a federate. Then, some of the other federates are notified of this occurrence, and there is, possibly, a response required of those federates, which may be communicated to some other set of federates.

The two "global" processes serve different purposes in the specification. FedExists represents a constraint that must be maintained globally by the federates working together. It specifies that the execution must be created exactly once, and that there must be no activity unless the execution exists. This property can not be guaranteed by any single federate, so it is not appropriate to be placed in the SimInterface. It represents a recognizable constraint that must hold at all times during the protocol, so it should not be spread out across the mini-protocols. FedExists localizes this constraint in the same way that FedJoined localized its constraint for an individual federate.

The second "global" process, WhatSims, rather than representing a constraint that must be maintained by each of the components, represents an element of state that must be kept outside of any federate. Each of the mini-protocols depends on this information: No message must ever be sent to any federate that is not joined. By separating this concern, we can simplify each of the mini-protocols since they don't have to maintain this state on their own.

WhatSims contains a feature that might be somewhat surprising: an event that is not associated with any federate. Usually we think of events as representing messages in the interaction of the software system. In this case, the event checkSims occurs but does not have any corresponding message in the actual system. This is indicated by there being no prefix on the event even though it occurs in the **Glue**. This is a standard CSP technique for broadcasting state information within a structured specification. In CSP, these events are typically hidden. In WRIGHT this is handled automatically as part of the configuration process, because after instantiation the renamed event will never occur in the alphabet of any other process.

6.4 Analysis of HLA using the WRIGHT specification

In the previous section we showed how the architectural structure of WRIGHT helps us structure our formalization of HLA, and how we can describe both the federate interface and the RTI using collections of simple, traceable processes. By abstracting from the details present in the IFSpec and separating different concerns, the specification supports an incremental approach to formalizing a complex architectural style.

In this section we will present examples of the analyses that we performed using WRIGHT as part of the development of the HLA standard. We will show how these insights are enabled by the formalization and how the structure of the WRIGHT specification ensures that our insights can be tied to properties of the IFSpec itself.

Our first two examples represent discoveries based on the use of an automated analysis tool. This tool is based on a commercial tool, FDR [FDR93]. FDR provides a means of performing refinement tests on CSP. In particular, it is capable of detecting potential deadlocks in CSP processes. Our tool converts a WRIGHT specification into a collection of CSP processes that can be checked using FDR. This permits us to locate and describe potential execution traces of a federate that would lead to conflicts, such as violation of preconditions and inconsistent states in different federates (or in the RTI). Automation of WRIGHT analysis is discussed further in chapter 7.

The third and fourth examples are of a slightly different kind. Rather than representing discoveries that result from analysis of an existing specification, they are discoveries that we made during the process of formalization. Our examples show how, by attempting to make an informal description precise, ambiguities or omissions in the original document are revealed.

Before proceeding with the examples it is important to note the following: For expository reasons our WRIGHT extracts will not be exact renditions of the behavior specifications as written in the full specification. Instead, we will freely use a rewriting that has the same behavior but more simply illustrates the point we are making. These recastings are equivalent processes, and the automated tools find the behavior in the original. We have simply restructured them, based on the discoveries of the tool, to make it easier for the reader. We did not need to "guess" this representation in order to make the discoveries that we did.

6.4.1 Creation of the Execution

Our first example discovery rests in the start up behavior of a federation execution. A federate must, when starting, decide whether to create the execution, and then, before sending any other messages, it must join the federation. This is represented by the following extract of the SimInterface:

$$\textbf{Interface Type}\ \mathsf{SimInterface} = \mathsf{JoinFed} \sqcap \overline{\mathsf{createFedExecution}} \rightarrow \mathsf{JoinFed}$$
$$\quad \textbf{where}\ \mathsf{JoinFed} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{ContFed}$$

This extract shows how the first action of any federate is to decide (internally, represented by $\sqcap$) between an initial createFedExecution or joinFedExecution. The corresponding part of the RTI **Glue** is as follows:

$$\textbf{Glue} = \square\ i : 1..nsims \bullet \mathsf{Fed}_i.\mathsf{createFedExecution} \rightarrow \mathrm{WaitForSim}_{\{\}}$$
$$\quad \textbf{where}\ \mathsf{WaitForSim}_S = \square\ i : 1..nsims \bullet \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \mathsf{WaitForSim}_{S \cup \{i\}}$$

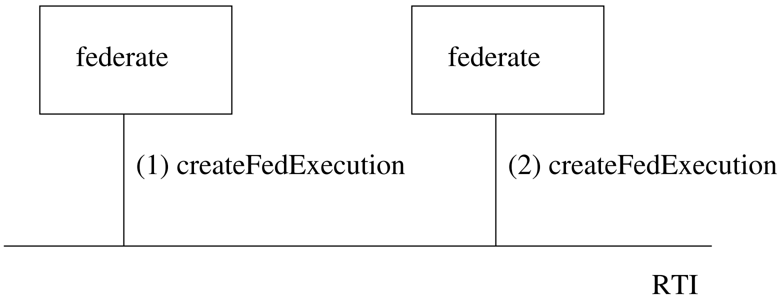

Figure 6.5: Oops! Deadlock when two federates create.

This specification states that the first event must be a createFedExecution from any one of the federates. After this message is received, the RTI **Glue** is in the state WaitForSim, in which it is possible for any of the federates to send joinFedExecution. Note how after createFedExecution the process' control state changes (to WaitForSim) but after joinFedExecution it stays the same (although the data state changes). This indicates that there must be exactly one create, but there can be many joins.

Trouble arises with the trace represented in figure 6.5. Each federate has to make the decision about whether to create internally, without any information from outside itself. If the execution has not been created, then it is not permitted to join, but if it has been created, it must join. This problem is detected as deadlock between the RTI and the second federate. It is also detected as a deadlock with the first federate, because it may choose to join without creating.

By formalizing the specification in WRIGHT, this problem is detected immediately and automatically. This problem also exists in the official standards document, because it defines a precondition, that a federate must not create the execution if it already exists, but does not provide any way for a federate to discover the information it needs to satisfy the precondition. Recall that the interface is intended to be the only communication mechanism between federates, and so the omission of this facility in the interface is a serious problem.

The structure of the WRIGHT specification leads us directly back to the source of the problem in the IFSpec. In its description of the createFedExecution, the IFSpec states "The named federation execution does not exist" as a precondition. The message joinFedExecution has a corresponding precondition "The named federation execution exists." Thus, the WRIGHT structure described is directly traced to the informal specification. What the IFSpec does *not* state is how a federate discovers whether the execution exists or not.

Because WRIGHT structures an interaction into roles and glue, the specification must take into account the point of view of a single federate. The general IFSpec document, on the other hand, does not make this distinction clean, and so sometimes it fails to account for global knowledge, available to an omniscient observer, that is not available to a single federate.

6.4.2 Paused on Join

In our previous example, we saw how the WRIGHT analysis reveals potential problems in the IFSpec. By locating a deadlock in the formal specification and providing an example scenario in which it might occur, the analysis tools pinpoint trouble spots in the informal documentation.

For the particular example above, in which deadlock occurs immediately, or after at most two events, it might be argued that this isn't a very deep insight; any development effort could not get very far without stumbling across this situation. Perhaps it isn't worth the effort of formalizing the specification, since the problem would have been found and solved anyway. This is not true for two reasons. First, it is not unreasonable to believe that an implementation effort, such as a prototyping effort, would simply solve the problem in its own system (perhaps by selecting one of the federates to be the creator) and continue without ever realizing it was an intrinsic part of the specification. To our knowledge, none of the prototyping efforts raised this issue as a problem with the interface specification. Thus, it might only be recognized as a problem after many of these systems have been built and people start to realize that every one of them had to develop their own, possibly different, workaround. In an informal specification direct to implementation strategy, the multiple creation problem might only become significant, and thus problematic, when efforts are made to integrate separately developed federates, which would occur relatively late in the prototyping life-cycle. Through WRIGHT it has been detected earlier, before the interface is published as a full standard on which procurement decisions are based.

The second reason that this insight should not be dismissed as trivial is that it only represents the simplest example of an entire class of problem that can be located by the automated tools. Consider now the following extract of the WRIGHT SimInterface specification:

$$
\begin{array}{l}
\mathsf{JoinFed} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{ContFed} \\
\mathsf{ContFed} = \overline{\mathsf{requestPause}} \rightarrow \mathsf{ContFed} \\
\qquad\qquad \sqcap\ \mathsf{WaitForEvent} \\
\mathsf{WaitForEvent} = \mathsf{schedulePause} \rightarrow \overline{\mathsf{pauseAchieved}} \rightarrow \mathsf{FedPaused} \\
\\
\mathsf{FedPaused} = \overline{\mathsf{requestResume}} \rightarrow \mathsf{FedPaused} \\
\qquad\qquad \sqcap\ \mathsf{PauseWait} \\
\mathsf{PauseWait} = \mathsf{scheduleResume} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \mathsf{ContFed}
\end{array}
$$

This extract focusses on the pause and resume behavior of a federate. It indicates that in the ContFed state, the federation is "running." That is, it can carry out normal events (not shown), it is permitted to request a pause, and it should expect the possibility that a pause may be scheduled. Once a pause is scheduled, the federate pauses itself, notifies the RTI of its success, and is then in the state FedPaused. This is the inverse of ContFed — in this state, it does not carry out normal events, but instead may request a resume (but not another pause), and should expect that a scheduleResume will occur. Once it does, the federation is running again.

The RTI glue shows how these events are combined in different federates through mini-protocols such as HandlePause:

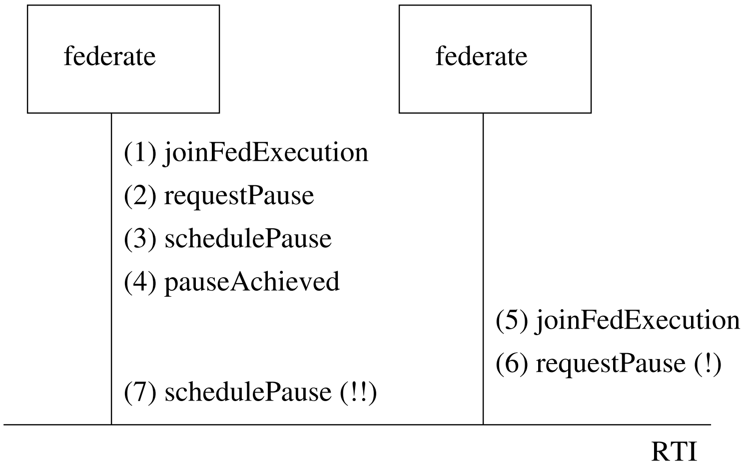

Figure 6.6: Another deadlock: Federates are confused about pausing.

$$\mathsf{HandlePause}_S = \Box\, i : S \bullet \mathsf{Fed}_i.\mathsf{requestPause} \rightarrow (;\, i : S \bullet \overline{\mathsf{Fed}_i.\mathsf{schedulePause}} \rightarrow \S) \; ; \; \mathsf{HandlePause}_S$$

HandlePause indicates that whenever a federate requests a pause, all federates will receive a notification via the message schedulePause. Corresponding mini-protocols (not shown) handle resume requests and recognize pauseAchieved and resumeAchieved to keep track of whether the federation as a whole is paused or running.

This protocol of pause and resume results in a problem as depicted in figure 6.6. Deadlock arises because after event (6), Fed_2.requestPause, the next event according to the RTI will be Fed_1.schedulePause, but according to Fed_1 it must be Fed_1.scheduleResume. The problem with this sequence is that when it joins the federation execution, Fed_2 *doesn't know the system is paused*.

What is worth noting about this scenario is: It requires enough complication to be difficult to locate by reading the informal documentation. It would be even more difficult to locate this problem by executing prototype implementations, since a normal execution of a federation would involve many more messages than just those for joining and pausing. Under normal operation, the join-then-pause behavior is a race condition between Fed_2 joining and Fed_1 pausing, which would make it even more difficult to detect through trial-and-error. The WRIGHT tools found this property even though we weren't looking for it in particular and we didn't know it was there.

Once we have found it, however, the WRIGHT specification leads us back to the IFSpec document and the source of the problem. The message joinFedExecution contains a returned parameter "federation state information (to be defined later)." Our analysis indicates that whether the system is paused or not *must* be included in this information in order for the federate to obey its constraints.

6.4.3 What's in NotPausedEvents?

In this section we explore another kind of insight that can be gained through formalizing an informal architecture specification. Sometimes a document contains ambiguities or omissions that must be resolved before a formal specification can even be attempted, before there is any way of exploring the properties of the formalization.

While the previous examples showed how issues could be raised through the use of automated analysis tools, in this and the next section we explore how the discipline required to formalize what had been informal can lead us to other kinds of insights.

Recall the process ControlPause from our earlier definition of SimInterface:

$$\mathsf{ControlPause} = \mathsf{RUN}_{NotPausedEvents}\ ;\ \overline{\mathsf{pauseAchieved}} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \mathsf{ControlPause}$$

This process controls what a federate is permitted to do during a pause, and what messages it can expect to receive during a pause. Recall how RUN_S in a process specification indicates that any of those event may occur without constraint until the event following RUN happens. After that, there are constraints on what can occur until the state RUN re-occurs. Thus, ControlPause indicates that none of the events in the set $NotPausedEvents$ is permitted during a pause (*i.e.*, after a pause is achieved but before a resume).

The issue this specification raises is a follows: What events are in the set $NotPaused$-$Events$? Intuitively, we would expect quite a few events. If a pause means that the simulation is not advancing, then it doesn't make sense for a federate to continue with object attribute updates, time advance requests, *etc*.

However, according to the documentation as written, only requestPause and schedulePause are in the set $NotPausedEvents$. That is, the only events that are affected by a pause are those directly relating to pausing. A pause has no concrete effect on the execution of the simulation at all! This is certainly worth noting about the IFSpec, since it is so unexpected. The structure of that document (using pre- and post-conditions to represent constraints from states of the system) makes it difficult to recognize this; the events directly related to pausing discuss "the federation is paused" and "the federation is advancing," so one expects that this will have some effect on other events, but it isn't mentioned anywhere else. By attempting to make precise the concept of a pause, mentioned in the IFSpec, we realize just how imprecise it is.

This observation gets at a more subtle kind of discovery than the deadlock examples above, in that it shows something one discovers simply by trying to formalize the informal. It isn't that we wrote the specification down and then analyzed it, it's that we couldn't write down what we expected because the necessary information wasn't present at all.

This kind of discovery is important to a standards effort because it highlights potential points of divergence between different implementations as well as situations where the intentions of the designers have not been made explicit, and thus could be misinterpreted by users of the specification.

6.4.4 Exceptions

Another issue we discovered through the process of formalization is the handling of exceptions. In the initial IFSpec, it wasn't clear whether the exceptions mentioned in the document indicated messages that could be signalled or whether they were simply signposts for the implementer of a federate or RTI. Some possible alternatives regarding how the RTI could handle one protocol, that for resignation of a federate from the execution, are shown in figure 6.7.

$$\mathsf{HandleResign}_S = \square\, i : S \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{HandleResign}_{S-\{i\}}$$

$$\begin{aligned}\mathsf{HandleResign}_S = {} & \square\, i : S \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{HandleResign}_{S-\{i\}} \\ & \square\, (\square\, i : (1..nsims) - S \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \\ & \qquad \rightarrow \overline{\mathsf{Fed}_i.\mathsf{notJoinedException}} \rightarrow \mathsf{HandleResign}_S)\end{aligned}$$

$$\begin{aligned}\mathsf{HandleResign}_S = {} & (\square\, i : S \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{HandleResign}_{S-\{i\}}) \\ & \square\, (\square\, i : (1..nsims) - S \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{HandleResign}_S)\end{aligned}$$

$$\mathsf{HandleResign}_S = \square\, i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{HandleResign}_{S-\{i\}}$$

Figure 6.7: Some alternative representations of HandleResign.

The first alternative "ignores" the exceptions, describing the protocol as it occurs when pre- and post-conditions are obeyed. Only federates that are currently joined (represented by the set S) are permitted to resign. This representation views exceptions as "bad things that might happen;" the specification describes only valid behavior, and analysis using the standard tests will discover if it is possible for a federate to disobey this precondition. If a federate might resign when it hasn't joined, deadlock will occur, since HandleResign will refuse the resignFedExecution event in this case. If this can never occur, then no deadlock is possible, and the exception will never arise.

The second alternative also exposes valid versus invalid behavior, but by highlighting the invalid behavior as an explicit exception message notJoinedException. This specification differs in two ways from the previous. First, it requires that the specifier predict this possibility ahead of time and explicitly recognize it as a possibility. The first approach simply describes the desired behaviors, whereas to be effective the second must predict and cover all deviations from that behavior as well. Second, unlike the first approach, the second indicates what will actually occur after the exception. In this case, the federation continues as if the invalid resignation had not happened, indicated by the return to $\mathsf{HandleResign}_S$ without any change in state.

Although the first alternative specification does have an actual behavior if an invalid resignFedExecution is attempted, deadlock, this can't be seen as defining the actual behavior of any RTI implementation. In WRIGHT, deadlock is used simply to indicate some kind of problem in the specification and cannot be taken to mean that the system will actually halt. For example, in a networked message passing environment, a dropped message, where the receiving party must be waiting for a message but isn't, would be detected as deadlock, but in an implementation it would result in some unpredictable, not-quite-right, behavior, not in an immediate system lock-up.

By using explicit exceptions in the specification the second alternative has clarified how an RTI is expected to behave if this bad thing does happen, but it does so at the expense of making it more difficult to detect the problem during *a priori* analysis. The analyst must predict the problem and explicitly construct a check for the possibility of a notJoinedException event, rather than depending on the standard deadlock test.

The third and fourth alternatives have been included to illustrate another point: Because of the event oriented nature of a WRIGHT behavior specification, if one permits undesired behaviors, one must use events to flag them or they can not be detected during analysis. The third example, like the second, describes two behaviors for the RTI, one normal and one exceptional. If a joined federate resigns, fine; the federate is removed from S. If, on the other hand, a non-joined federate attempts a resign, this behavior is described by the other branch of the choice: In this case, the event is ignored.

In effect, this third alternative treats a non-joined resignation as a *legal* behavior by a federate. A test for illegal activity will not detect this behavior, since it is not distinguished by either events or refusals (*i.e.*, the inability to perform an event) in the process. This is made clear by noting that the third and fourth alternatives are in fact equivalent processes. It is just that on the surface the third *appears* to distinguish the two cases, when in fact it treats them identically.

The point here is that this WRIGHT specification is undertaken for a specific reason: to facilitate the analysis of the HLA and to detect potential problems in the IFSpec. If we are to accomplish this goal, we depend on the semantics of the specification and how it relates to the tools we have available. A WRIGHT specification is not just a piece of documentation to give a sequence-of-events view of the computation; it carries a precise, powerful semantic model that is used to locate problems and to prove properties of the systems to be constructed.

6.5 Review of Techniques

In the previous section, we saw how the WRIGHT specification of HLA led to the exposure of important issues in the interface specification. Through the exercise of formalizing what had been informal, it became clear that there were many ambiguities in the specification, such as the effect of a "pause" on the execution, and that several concepts, including "exceptions," used in the interface specification document were not fully specified as to how they would be realized in an implementation.

We also showed how the formalization in WRIGHT, by providing a standard set of consistency checks, led us to locate potential trouble-spots in the protocol described. An automated test for deadlock in a connector led us to two examples of problems in the specification, one involving creation of the execution and the other related to the interaction between start-up and pausing.

In this section we will revisit the specification with a slightly different view. We will consider how specific techniques used in this specification can be useful in general in the formalization of a system in WRIGHT. We will consider techniques at two levels of description: first, those related to the architectural structuring of systems, and second, those related to the specification of behaviors in WRIGHT's CSP notation.

6.5.1 Architectural Structure

WRIGHT provides mechanisms for making architectural distinctions that must be considered when developing an architecture. In this section we discuss three of those: The distinction between a single system and a family of systems, the distinction between components and connectors, and the distinction between different parts of those architectural elements.

The first distinction we will discuss is between a single system, represented in WRIGHT as a configuration, and a family of systems, represented as a style. Depending on which of these is to be considered, the issues are different and the elements to be described are different.

For a style, the issues relate to the kinds of elements (to formalize as types) and the constraints that will be applied to the use of those elements. These decisions are based on the properties that we want the systems to share, the analyses that will be important, and the kind of infrastructure support that are to be developed.

A single system, on the other hand, leads us to look for the elements of an architectural configuration, individual computation boundaries (components), kinds of interactions (connectors), and specific pathways of communications (attachments).

In HLA, we recognized that the description was of a family of systems, covering more than one potential configuration. This led us to consider the federates uniformly, in terms of a shared interface type, and to look for a description of the common elements that will span systems. In this case, the composition of federates via the RTI led us to a single connector type with the constraint that all communication is brokered by a single connector instance.

Other specifications (say of a prototype federation implementation) might be recognized as describing a single system, which would lead us instead to consider the actual pathways of communication and the specific computations. In this case, the architectural specification would distinguish different federates based on the actual computations that they carry out, and consider the specifics of how they communicate.

A second question is how to divide the elements of the architecture into distinct entities. For example, when do we distinguish different component types from each other, and does a given part of the specification describe a component or a connector?

One consideration in viewing a specification of a style is that, typically, the component types are taken as constraining but not fully defining the components that will appear in a system, while the connectors types are viewed as abstract definitions of a fixed pattern of interaction. While there are exceptions (*e.g.*, reusable component frameworks and data-dependent connectors), it is typical for a configuration that instantiates a style to *use* a set of predefined connectors, while it *defines* a set of instance-specific components. That is, we think of an architectural style's connectors as defining a set of operators over the components that are provided by a configuration.

The definition of RTI as a connector fits into this pattern, as it provides a means of composing simulation fragments (federates) into a complete simulation. We recognized that the abstraction presented by the IFSpec to a developer was that of a set of *interacting federates* rather than of federates interacting with the RTI. This led us to describe the RTI behavior as the glue of a single connector, and to specify the relations between different federates' messages as a first class entity in the architecture.

As an alternative to treating it as a connector, one might have considered describing the RTI as another kind of component in addition to the federates. In this case, the SimInterface would have been a simple "bundle-of-messages" type connector, and the details of how these messages are combined would be hidden inside the RTI component specification. While this would, perhaps, clarify the task faced by an RTI implementer, it hides the relationships between different federates that someone constructing a complete simulation would need to understand.

This kind of distinction is an essential element of the architect's task — to select the set of abstractions that both provide the properties that are desired and help other developers to understand and reason about systems they are developing. In our case, the developers of the HLA made these decisions based on the simulation domain; they present an abstraction of interacting federates, and the WRIGHT specification reflects that.

A third issue when specifying an architecture is how to structure the description of each of the identified component and connector types. When formalizing a connector, WRIGHT's architectural structure guides our analysis. By looking for the *roles* in the specification, we can structure the specification to take into account different points of view. In the RTI, each individual federate, described as a role, is one point of view, and the overall computation, coordinated by the glue, is another.

A similar, but more commonly recognized, structuring of an architectural entity is the distinction between the computation and the interface of a component. WRIGHT requires *all* interactions to be specified, as ports, rather than only the primary interaction. This contrasts with, for example, an object specification, where only the incoming method calls are specified in the interface, ignoring out-bound messages and global data references. Specifying all interactions helps ensure that a component can be understood in isolation and that it is clear what information is available to it.

When specifying a system, it is easy to forget the different points of view in the system, and what information each potential computation has or does not have. The problems detected in the HLA regarding creation of the execution is an example of a point-of-view problem; the draft specification provides a precondition on the create message that refers to a *global* concept – whether the execution exists. It is only through analysis that we could determine whether this information is available locally – in an individual federate.

6.5.2 Behavior Specification

In the remainder of this section we will describe techniques that we can use to structure the description of behaviors in WRIGHT. There are a number of challenges to describing and analyzing the behavior of an architectural style, several of which we faced in the HLA case study. In this section we discuss how our use of a formal model, CSP, as the basic notation for WRIGHT provides techniques that we can apply to the task of architectural description. We will discuss four particular CSP specification techniques that we can apply in WRIGHT: Mini-protocols, macro-processes, selective omission, and internal state processes.

One kind of challenge is dealing with the gap between the informal and formal specifications. When attempting to develop a formal specification, it may be the case that the form of the informal description does not match the notation of the formal description. We must find ways of representing the concepts of the informal specification in the notation to be used in the formal specification. In our case, the messages were described in the IFSpec using pre- and post-conditions and exceptions. WRIGHT provides a process algebra (patterns of traces) model of behavior. There may be an *impedance mismatch* between the forms of the description.

At the same time as we shift from the informal to the formal, we must be able to get from the formal specification back to the informal, to capitalize on any insights gained. There must be *traceability* between the specifications. While the change in notation may ease analysis, the insight gained about the formal description must be brought back to the informal description so it can show how the actual abstractions or documentation can be improved.

Another kind of challenge has to do with handling the scale and level of detail in a complex specification. The scale of the system being described may make a full, exhaustive, formalization prohibitive; analytic results must be obtainable with a partial specification. At the same time, while describing only part of the system, or some aspects of the system, we must not invalidate our results by ignoring an interaction or the possibility of expanding the description to cover more issues.

Thus, we seek an *incremental* description, meaning that we can gain analytic leverage with a minimum effort and at the same time our initial effort also contributes to an eventual, more complete description. As we cover more of the details of the system, each new description must add to the whole at only a small additional cost.

Additionally, we want the formal description to raise the level of *abstraction* at which we consider the system. The intent of analysis is to focus on those aspects that are of paramount importance, making them tractable, while suppressing the complexities that can be considered separately. Thus, even if a formal description covers every part of the system, it does so by highlighting certain aspects of the system, rather than exhaustively recording every possible property of it.

In order to achieve these goals, we use four techniques to structure our behavior descriptions in WRIGHT:

1. *Mini-protocols* encapsulate one distinct aspect of the overall protocol in a separate, identifiable process.

2. *Macro-processes* aggregate collections of events into building blocks so that the collection can be treated as a single unit.

3. *Selective omission* of details permits us to keep the description tractable.

4. *Internal state processes* record pieces of information that are needed in many parts of the description but can be maintained locally.

The first three techniques can be illustrated by one of the processes that form a part of the SimInterface interface type. Recall the definition of FedJoined:

$$\mathsf{FedJoined} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{RUN}_{FedEvents}\ ;\ \overline{\mathsf{resignFedExecution}} \rightarrow \S$$

Mini-protocols

FedJoined is one of many *mini-protocols* in the WRIGHT specification of HLA. It describes only one small aspect of the behavior of a federate, that of joining and resigning from the execution. Other examples of mini-protocols in SimInterface are ControlPause, which describes pausing, and ObjMgmt, which describes what messages are used to control attributes on individual objects. Mini-protocols are also used in the RTI **Glue**. We showed one example in the discussion of pausing and joining, HandlePause. In the RTI **Glue** there is a mini-protocol for each event that a federate initiates independent of other events and that results in RTI messages to other federates.

While each of the mini-protocols describes only a part of the total behavior, they combine to encompass all of the behaviors that are allowed by the interface specification. This is because the mini-protocols are tied together by shared events. FedJoined, for example, describes how all of the events in the set $FedEvents$ are related to joinFedExecution and resignFedExecution, but does not show how the members of $FedEvents$ relate to each other. The mini-protocol OwnMgmt shows how two of these events, reqAttrOwnDivestiture and attrOwnDivestitureNotifyRelease are related, while ignoring all behavior related to join and resign. By sharing events in their alphabets, the two mini-protocols together describe a larger behavior, combining both the join/resign behavior and the ownership request/response behavior, that can be analyzed as a whole.

Thus, mini-protocols provide a mechanism to achieve incrementality in the specification. By separating out a small portion of the total protocol, the effort to provide formal analytic leverage is reduced. By tying the mini-protocols together with shared alphabets, a larger specification can be built up from several smaller units.

The use of mini-protocols also supports traceability and provides a means of bridging the impedance mismatch. Each of the mini-protocols is derived from a small set of the pre- and post-conditions described in the source interface specification document.

For example, the message joinFedExecution has two pre-conditions, "The named federation execution exists" and "The federate ID is not joined." It has one post-condition, "The federate is a member of the named federation execution." The FedJoined process deals with a specific concept exposed in these pre- and post-conditions, membership in the federation. There are exactly two messages whose *post-condition* mentions membership in the federation, namely, joinFedExecution and resignFedExecution. These are the events that are called out in FedJoined as critical to this mini protocol. As described by the pre- and post-conditions, a federate is not a member either before a join or after a resign. Any event that is not permitted unless the federate is a member (as described by that message's pre-conditions) is prevented from occuring except between the join and resign.

The other pre-condition on join, regarding the existence of the execution, is controlled by a different mini-protocol, FedMgmt, shown in appendix B.

What we have done in this situation is to recognize critical *state* abstractions in the described pre- and post-conditions. Key phrases (*e.g.*, "is a member", "is not a member") indicate that a message changes the abstract state of the overall computation. Other messages are controlled by these states, as indicated by including references to them in their pre-conditions. The mini-protocols can then represent these critical states by collecting together the sets of events that are permitted in some states and denied in others. The controlling events (those that change the state, as indicated by references in post-conditions), are used to trigger changes in the processes that represent the mini-protocol.

Macro-processes

We use another technique, *macro-processes*, to represent the categories of events that are controlled by the critical states. In the example, note how the process RUN is used to indicate when events in the set $FedEvents$ are permitted; as we indicated, this is only while the federate is a member of the execution. The process RUN has an important property: While it permits a set of events to occur, it does not further constrain *which* events will occur, or *how many* events will occur. This is important because it means that we can indicate a period outside of which the events will certainly *not* occur, but leave it to other mini-protocols to indicate any constraints on how those events are selected or controlled. Thus, using $\mathsf{RUN}_{FedEvents}$ highlights $FedEvents$ as a general category of events that are constrained together by the mini-protocol. They can be lumped together here, while they will be considered separately elsewhere.

To use RUN this way, it is fit into a larger process in a particular way. First, there are the conditions that lead to the events being permitted. For example, the event joinFedExecution marks a transition from non-membership to membership. These events are prefixed to the RUN process: joinFedExecution$\rightarrow$RUN. Next, the conditions that lead to the prohibition of the events in the set are identified. In this case, it is the occurrence of resignFedExecution. These conditions are used as a guard on the process describing behavior after the events are no longer permitted, and this process is appended to the RUN using a semi-colon: RUN;resignFedExecution$\rightarrow$§.

We refer to this technique of bundling events as a macro because the form is somewhat analogous to the way in which macros in programming languages create a simple-appearing construct for a routine but relatively complex construction. When we use RUN in a process, e$\rightarrow$$\mathsf{RUN}_S$; f $\rightarrow$... we are bundling up a complex set of traces to appear somewhat like a single event in the mini-protocol: e$\rightarrow S \rightarrow$f $\rightarrow$...

Other macros besides RUN can be used in this manner. For example, the ObjMgmt group, shown in appendix B, uses a macro WaitForEvent that permits a *single* event from a set, rather than zero or more events.

Selective Omission

Another key technique in WRIGHT behavior specifications is the use of *selective omission* to simplify the specification and increase the level of abstraction. In the HLA specification, we have chosen to omit parameters to all messages whenever possible.

For example, the joinFedExecution message is specified as taking a simulation name, a simulation ID, a federation execution name, connection parameters, an a value of federation time when the join will occur. None of these affect the pattern of events that will follow, so we have left them off.

In other cases, for example, the subscribeObjClass method, the parameters do affect the details of the communication, but we have left them out anyway. In this case, we use *non-determinism* to indicate that a decision must be made, but that the details of that decision are not specified at this level of abstraction. This permits a conservative analysis of the RTI, in the sense that our specification permits more behaviors than are actually possible, so anything that is *not* possible in this specification is definitely not possible in a correct RTI.

In the case of subscribeObjClass, the parameters indicate what attributes of an object are of interest to the federate. This information would then be used by the RTI to control what attribute value updates are transmitted to that federate.

The protocol as originally specified indicates a collection of "discovery predicates" that define what values of what attributes are to be transmitted. When the predicates become true, an instantiateDiscoveredObj message is sent. Then new values are indicated by reflectAttrValues whenever a new value is shown, until the discovery predicate becomes false. At that point, a removeObj message is sent, informing the federate that it should no longer track that attribute (or object).

Instead of reflecting the full complexity of discovery predicates (which are not defined in the informal document), we have chosen to abstract away from this element of the informal specification. This simplifies the attribute distribution protocol considerably. One part of this protocol, the creation of a new object, was shown earlier. Another, more complicated portion, is the attribute update mini-protocol:

$$
\begin{aligned}
\mathsf{NewAttrValue} = \square\, i : 1..nsims \bullet\ & \mathsf{Fed}_i.\mathsf{updateAttrValues} \rightarrow \mathsf{checkSims?S} \\
& \rightarrow (\sqcap\, J : \mathbb{P}\, S \bullet (;\, j : J \bullet \overline{\mathsf{Fed}_j.\mathsf{removeObj}} \rightarrow \S); \\
& \quad (\sqcap\, K : \mathbb{P}\, S \setminus J \bullet (;\, k : K \bullet \overline{\mathsf{Fed}_k.\mathsf{instantiateDiscoveredObj}} \rightarrow \S); \\
& \quad (\sqcap\, L : \mathbb{P}\, S \setminus J \setminus K \bullet (;\, l : L \bullet \overline{\mathsf{Fed}_l.\mathsf{reflectAttrValues}} \rightarrow \S)))); \\
& \mathsf{NewAttrValue}
\end{aligned}
$$

The NewAttrValue protocol indicates that following an updateAttrValues message from any federate, the RTI makes three related choices of notifications. These three sets are J, the set of federates that will need to remove the object from their representations, K, those that will need to add it to their representations, and L, those that already know about the object and need the new attribute value. Each of these sets are chosen *non-deterministically* in this specification, indicating that the details of how this is calculated is omitted. We are able to do this for two reasons.

First, the traces of this specification are a superset of the actual behavior specified in the RTI, and it is fully non-deterministic in this situation. Any specification that more precisely describes the RTI behavior regarding these events will be a refinement of this specification. This means that most properties that we can prove about this specification will necessarily be true of the actual RTI behavior. The only exception is a property that describes the possibility, but not the necessity, of the RTI carrying out a particular trace. For example, it is possible in this spec for the RTI to chose to send a removeObj without a preceding instantiateDiscoveredObj. Because of the nature of CSP refinement properties, these exceptions are not ones that could be counted on by any external observer – a federate, for example. It is not possible to *force* the RTI to have any of the behaviors permitted by the non-deterministic choice that are to be prevented by a refinement.

Second, it is reasonable for us to increase the number of options in this particular instance (that of the specific attribute values and discovery predicates) because we aren't, in this specification, concerned with the exact calculations that a federate will make. Instead, we are concerned with the overall pattern of interaction, and in detecting situations where a federate would be unable to do the right thing; that is, we are free to assume that a federate will be able to handle a certain amount of variation, and we look for situations where it definitely can't handle what is expected (such as not being able to know whether or not to create). Thus, the exact object space isn't of direct interest and it can be left to a different specification to explore.

Another important kind of omission in this specification is that we haven't described any of the events in the Time Management group. We chose to do this for two reasons. First, it isn't necessary to include all events in order to discover properties about the specification. Our observations regarding the federation management messages (join, pause, *etc.*) hold regardless of the set of other messages available. Second, our use of mini-protocols with shared alphabets makes it simple to add these later if we choose to explore that part of the specification. There would need to be new mini-protocols to show how the time management events relate to each other, and they would need to be added to any sets involved in macros to indicate how they are controlled by the other events. For example, they would be added to $FedEvents$ because time can only be managed by a joined federate. If any of the time events were to have a controlling influence over the other events (which they don't happen to), then mini-protocols could be developed to further constrain the system.

Internal State Processes

The final technique for specifying behaviors that we will discuss here is the use of *internal state processes*. This was illustrated above in the process WhatSims:

$$
\begin{aligned}
\mathsf{WhatSims}_S = {} & \mathsf{checkSims!S} \rightarrow \mathsf{WhatSims}_S \\
& \Box\, (\Box\; i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \mathsf{WhatSims}_{S \cup \{i\}}) \\
& \Box\, (\Box\; i : 1..nsims \bullet\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \rightarrow \mathsf{WhatSims}_{S - \{i\}})
\end{aligned}
$$

WhatSims has two parts. One part observes interaction events, and alters the state of the process WhatSims. The other part permits mini-protocols to query this state through an event that is internal to the connector **Glue** (checkSims). The reason to separate out the state maintenance is to simplify the other sub-processes in the **Glue**, which no longer have to carry out the observation of events that are otherwise irrelevant to the mini-protocols. If this calculation were not separated, every mini-protocol would have to include the joinFedExecution and resignFedExecution events in their alphabets and maintain a set-of-processes state variable. This would add two alternatives (for join and resign) to *every* state in *every* mini-protocol. In short, the readability of the protocols would be much decreased, and the opportunity for errors would be magnified.

By maintaining the simplicity of the mini-protocols, we increase the traceability of the specification. It is easier to see how a mini-protocol relates directly to a small set of pre- and post-conditions if it is not intertwined with other concerns. By localizing the maintenance of a given global concept, we also improve the level of abstraction in the specification. The abstract concept is both more precise and more easily understood in its direct form.

6.6 Discussion

This case study has shown how formalizing an architectural style in WRIGHT can provide insight into potential problems with an informal description of the style. Our formalization effort found issues in two ways: First, by requiring the rigorous consideration of definitions and structuring the document in terms of patterns of events, we found ambiguities and omissions that needed to be resolved before the meaning of the style could be made precise. Second, once we had a formal description, we were able to find inconsistencies and weaknesses in the protocols through the application of automated analysis tools.

We have also shown, through the use of WRIGHT to support the development of a complex, industrial scale architectural standard, how WRIGHT formalizations scale to practical problems. By scaling, we mean not that we can construct larger and larger WRIGHT specifications, but that we can continue to provide significant, practical benefits to system developers for a reasonable effort as the size and complexity of the *problem* increases.

To accomplish this goal, we have shown how we can gain insight into a large-scale specification, the IFSpec of HLA, and how those insights can be tied to the specification in such a way that they can reasonably be expected to influence and improve the efforts to develop simulations in the future.

Chapter 7

Tools for WRIGHT Checks

7.1 Introduction

An important means of making the analysis of architecture practical is to automate that analysis because, (1) it reduces the effort required to perform the analysis, and (2) it can find problems or properties that a human analyst would have missed. We have developed tools both to perform most of the standard WRIGHT consistency checks and to provide access to a commercial tool for other system-specific analyses that can be described by the architect.

By automating the standard checks, we provide a mechanism, much like type-checking in a programming language, for locating trouble-spots in a WRIGHT specification. Once the architect has developed a candidate specification, he or she can run a tool on the specification which responds either with a guarantee that the consistency checks are satisfied or a precise indication of where the problem is in the specification.

7.2 Automating Standard Checks

We have automated all of the compatibility checks described in chapter 3 except initiator-commits and configuration conformity to a style's constraints. Our primary means for doing so has been to use the commercial specification-checker FDR [FDR93]. We have used FDR as the core of the consistency checking tool. That is, our tool generates input to FDR from a WRIGHT specification and then executes FDR on that input.[1] A summary of our handling of each test is given in figure 7.1.

7.2.1 FDR

One of the most successful forms of automated analysis for formal methods is exhaustive checking of a finite model [$B^{+}90$]. There are two kinds of exhaustive checker. One kind determines whether a given property holds in every state of a model, and the other determines whether two state machines are related in a particular way. Both kinds of tool

[1] As we will discuss in section 7.3, the architect can also use FDR directly to perform other analyses, using our tools to provide FDR-compatible versions of the WRIGHT specification.

- Handled via FDR:
  1. Port-Computation Consistency (component)
  2. Connector Deadlock-free (connector)
  3. Roles Deadlock-free (role)
  8. Port-Role Compatibility (attachment)
- Handled directly, or through another test:
  4. Single Initiator (connector)
  6. Parameter Substitution (instance)
  7. Range Check (instance)
  11. Attachment Completeness (configuration)
- Not currently handled:
  5. Initiator Commits (any process)
  9. Style Constraints (configuration)
  10. Style Consistency (style)

Figure 7.1: Overview of test automation.

exhaustively search the state space of a finite-state model and either conclusively determine that the check is satisfied or provide a counterexample (such as an execution of the state machine after which the property does not hold).

This technology is available for a variant of CSP through the tool FDR. FDR (which stands for "Failures, Divergence, Refinement") automates the test of whether one process refines another. Recall that a process P refines a process Q (written $\mathsf{Q} \sqsubseteq \mathsf{P}$) if the failures of P are a subset of the failures of Q. The refinement test provides strong analytic capability for CSP in two ways: First, it provides a guarantee of substitutability of one process for another, much like subtyping between abstract datatypes. That is, if a process P refines a process Q, then, in any context in which Q appears, P can safely be substituted for Q.

Second, many properties can be expressed as an abstract "specification" process, such that any process that refines the specification process is guaranteed to have that property. This is because refinement permits a process to guarantee both the presence of a given behavior and the absence of others. If a given trace is not to be allowed, then the specification process omits that trace, and no process that includes the trace will refine it. If a given trace is to be required, then the specification process omits all other options from its refusals, and any process that refuses to engage in that trace will not be a refinement.

For example, consider the property of deadlock-freedom, which we have used extensively in previous chapters, both as a consistency check on our types and more globally in the JC case study of chapter 5. We defined deadlock-freedom directly in terms of the traces and refusals of a process (a process is deadlock-free if it never refuses all events until after $\surd$ has appeared in its trace), but it can also be cast as a refinement test: A process P with alphabet A is deadlock-free exactly when it is a refinement of the process DF_A:

$$\mathsf{DF}_A = (\sqcap\ e : A \bullet\ e \rightarrow \mathsf{DF}_A) \sqcap \S$$

To understand this, note that this process permits all possible traces over the set A, *but* that it never refuses all events (the $\sqcap$ operator always picks exactly one of the events). It does permit processes to accept more than one event, because that simply reduces the number of refusals.[2]

Thus, if we can translate WRIGHT specifications into the input format of FDR and convert a property of the specification into a refinement test, then FDR can be applied to confirm or deny that property for our specification.

7.2.2 Refinement Based Tests

Several of our tests have been directly defined as refinement tests, and so FDR's refinement check is directly applicable to their automation: Port-computation consistency and port-role compatibility (defined on pages 72 and 77 respectively). Further, as we described above, deadlock-freedom can be framed as a refinement test, and thus both connector deadlock-freedom and role deadlock-freedom (p. 73) are also clearly candidates for automation via FDR.

[2]As long as the number of traces is not increased, refusals and acceptances are inversely related: more acceptances, fewer refusals.

There are three main complications to using FDR for this purpose: (1) ensuring that the specifications to be checked are finite-state, (2) calculating $det(P)$, the deterministic version of P, and (3) calculating the alphabet of a process. Finite-stateness is an intrinsic limitation of exhaustive checking technology. Calculating det is necessary because several of the WRIGHT tests require it for the construction of one of the processes to be compared. Alphabets must be calculated both because several of the checks use it directly and because FDR uses an "alphabetized" version of the $\|$ operator. We'll come back to these latter questions after a brief discussion of the need for finiteness.

7.2.2.1 Exhaustive checking requires a finite-state model

A key limitation of automatic exhaustive checking of models is that the models are required to be finite-state. This raises two concerns: First, whether the limitation to finite-state processes is too restrictive a limitation, and second, how to deal with the possibility that some specifications may not be finite-state.

Even when limited to finite-state process descriptions, WRIGHT provides an expressive notation for characterizing architectures. In particular, it is important to note that finite-state CSP processes can have infinite traces. For example, the process $\mathsf{P} = \mathsf{e} \rightarrow \mathsf{P}$ has an infinite number of traces, but only one state. Indeed, quite complex processes can be described within the finite-state limitation. For example, all of the descriptions in our three case studies are finite-state.

However, any given WRIGHT specification may not be finite-state. For example, subscripting can lead to a problem as follows:

$$\begin{aligned}
&\mathsf{P}_0 = (\mathsf{plus} \rightarrow \mathsf{P}_1) \,\square\, \S \\
&\mathsf{P}_i = (\mathsf{plus} \rightarrow \mathsf{P}_{i+1}) \,\square\, (\mathsf{minus} \rightarrow \mathsf{P}_{i-1}) \\
&\quad i > 0
\end{aligned}$$

This process has a state P_i for every non-negative integer.

Permitting infinite-state specifications would not be a problem if we could automatically determine whether a given specification is finite- or infinite-state. In that case, we would simply provide three answers to the check instead of two: Yes, the check is passed; no, there is a problem; or don't know, the specification is not finite-state. Unfortunately, it is not decidable in general whether a process is finite- or infinite-state. Thus, if we want to automate a check, we have two options.

First, we can restrict the syntax of WRIGHT to a subset known not to permit infinite states. For example, one possible limitation is to omit all subscripts and prohibit the recursive use of ";" and "$\|$." By restricting the process notation to exclude these constructs, the automated checks can be guaranteed to terminate.

Second, we can leave the syntax unrestricted and accept that we can't guarantee that our checks will terminate. We have chosen this latter option because we feel that it is important to provide as expressive and flexible a notation as possible. If architects need to be sure that the tools will apply, then of course they can choose to use the restricted subset that is guaranteed to be finite state. This will involve making tradeoffs in the description between expressing the full details of the architecture and being able to use the automated tools.

Finite but too big?

The inability of an exhaustive property checker to handle infinite-state systems is a *theoretical* limitation. There are other, *practical* limitations, however: (1) the model's size may be too large to construct within a given memory limitation, and (2) the amount of time required to exhaust the state-space may be unacceptable.

This limit means that many analyses are not practical to attempt to automate for state-spaces of the size and complexity of a realistic software system. However, by taking advantage of the architectural structure of a specification, WRIGHT is able to localize checks and thus keep the size of the models to be checked relatively small, even as the overall size and complexity of the system increases. Thus, we use the architectural structure of the specification to control the cost of automating analysis.

For example, the Message connector with a size 4 buffer has 5 states. An S-R component with one input and one output has 2 states. Rings of S-R components, (all deadlock-free) have the following numbers of states that must be traversed in order to exhaustively check deadlock-freedom:

# comps.	# states
1	2
2	8
3	38
4	192
5	1447

7.2.2.2 Calculating $det(P)$

In addition to the general challenge of controlling the size of the state space, we must address two specific issues for WRIGHT's checks: Calculating a deterministic form of a process and finding the alphabet of a process.

Recall the definition of $det(P)$ from chapter 3:

> *For any process* $\mathsf{P} = (A, F, D)$*,* $det(\mathsf{P}) = (A, F', \emptyset)$ *where* $F' = \{(t, r) \mid t \in Traces(\mathsf{P}) \land \forall\, e : r \bullet t ^\frown \langle e \rangle \notin Traces(\mathsf{P})\}$.

The reason that we are interested in this process is that it is useful for limiting the traces of another process. It defines a set of traces *without defining how they are selected*. Thus, if we place $det(\mathsf{P})$ in parallel with another process Q, we know that there will be no traces not in P, but that we haven't otherwise restricted the refusals of Q.

There are two ways in which $det(P)$ can be calculated. First, if P meets certain syntactic constraints, then it can be calculated by simply replacing all occurrences of the non-deterministic choice operator $\sqcap$ with the deterministic operator $\Box$. This is simple and efficient, and it works under many circumstances. The current version of our tool uses this technique.

The restriction on when this technique can be applied is that no processes that are combined via choice share any initial events. For example, consider two possible processes:

$\mathsf{P} = \mathsf{e} \rightarrow \mathsf{P} \sqcap \mathsf{f} \rightarrow \S$
$\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \sqcap \mathsf{e} \rightarrow \S$

Process P can be modified by simply replacing $\sqcap$ with $\Box$:

$det(\mathsf{P}) = \mathsf{e} \rightarrow det(\mathsf{P}) \,\Box\, \mathsf{f} \rightarrow \S$

With Q, however, this will not work: a simple substitution of $\Box$ for $\sqcap$ gives this:

$\mathsf{Qwrong} = \mathsf{e} \rightarrow \mathsf{Qwrong} \,\Box\, \mathsf{e} \rightarrow \S$

The problem is that when both sides of an $\Box$ expression have the same event, CSP defines it to be *non*-deterministic. That is,

$\mathsf{Qwrong} = \mathsf{e} \rightarrow (\mathsf{Qwrong} \sqcap \S)$

To get a deterministic process, we must *first* move e and *then* change the $\sqcap$ to $\Box$:

$det(\mathsf{Q}) = \mathsf{e} \rightarrow (det(\mathsf{Q}) \,\Box\, \S)$

While the transformations are usually fairly straightforward, we have not implemented a fully general version because FDR calculates $det(\mathsf{P})$ internally as a side effect of calculating the state graph. Although it isn't available to specifiers in the current version of FDR, it should be made available soon (according to FDR's developers).

7.2.2.3 Calculating alphabets

In addition to $det(\mathsf{P})$, our tool must also calculate the alphabets of all CSP processes it uses. There are two reasons for this: First, several of our checks use it directly. Second, FDR uses an *alphabetized* version of the $\|$ operator. That is, rather than simply specifying $\mathsf{P}\|\mathsf{Q}$, FDR requires that the operator indicate what events are synchronized: $\mathsf{P}\|_A\mathsf{Q}$ for some set of events A. The difficulty is that CSP as originally defined does not specify how the alphabet of a process is to be determined. It is generally assumed that either the alphabet will be "obvious" or that it will be explicitly and separately defined. WRIGHT takes the alphabet of a process to be exactly those events that are named in its definition. To understand this, consider the following processes:

$\mathsf{P} = \mathsf{e} \rightarrow \mathsf{f} \rightarrow \mathsf{P}$
$\mathsf{Q} = \mathsf{f} \rightarrow \mathsf{g} \rightarrow \mathsf{Q}$

What are the alphabets of P and Q? Because P names events e and f, they are assumed to form the alphabet of P. Similarly, Q names f and g, so they are the alphabet of Q.

The alphabet of a process affects its behavior in one major way: when it is composed via $\|$ with another process. What is the behavior of $\mathsf{P}\|\mathsf{Q}$? The alphabets of P and Q share the event f, so they synchronize on that event. On the other hand, events e and g are not shared, so the two processes do not synchronize on those events.

FDR has taken a different approach to alphabets. Rather than calculating the alphabets of a process and then using them to find the meaning of $\|$, FDR requires that the synchronizing events be explicitly named whenever $\|$ is used: $\|_A$. So we would declare $\mathsf{P}\|\mathsf{Q}$ in FDR as follows: $\mathsf{P}\|_{\{\mathsf{f}\}}\mathsf{Q}$.

Thus, our tool must calculate the alphabets of all processes that will be combined using $\|$. Fortunately, this is straightforward. The only complication occurs when processes are defined with recursion. For example, consider the alphabets of P and Q in:

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

P is capable of engaging in events e and f, so its alphabet includes both those events. Q, on the other hand, can engage only in the event f, so if it is used directly (*i.e.*, not as part of P), its alphabet includes only f. Thus, we can not find the alphabet in a *purely* syntactic manner: although P names only e directly, it includes f in its alphabet.

Fortunately, the alphabet of a process can be found by the simple expedient of calculating the events explicitly named in defining expressions (e for P and f for Q, above), and then finding the closure over the processes that are named by that process. In other words, we start with the set of immediate events for every process. Then, for each process we add to that set the event sets of all the processes named by the given process. Process P above names Q, so we would add its events to P's events. Q names no other processes, so its set does not change. The new set is all events named with at most one level of indirection. Then, we repeat the addition of events until none of the event sets have changed on an entire iteration, indicating that the set is the transitive closure of the original set. This will terminate because there must be at most a finite number of indirections that can be taken without reaching all process definitions, or the process will not be finite state. If the process is not finite-state, we don't care what the alphabet is, since FDR will fail in any case (as we discussed above).[3]

7.2.2.4 Example of FDR use

We now illustrate our use of FDR by showing how we would check compatibility for a simple port and role. Consider the following two definitions:

Role $\mathsf{DataUser} = \overline{\mathsf{get}} \rightarrow \mathsf{DataUser} \sqcap \overline{\mathsf{set}} \rightarrow \mathsf{DataUser} \sqcap \S$
Port $\mathsf{Reader} = \overline{\mathsf{get}} \rightarrow \mathsf{Reader} \sqcap \S$

These are encoded in FDR's ASCII notation as follows:

```
DataUser = ((get ->DataUser) |~| ((set ->DataUser) |~| TICK)
Reader = ((get ->Reader) |~| TICK)
```

In order to determine if these are compatible, we must verify that:

$$\mathsf{DataUser}_{+(\alpha\mathsf{Reader}\setminus\alpha\mathsf{DataUser})} \sqsubseteq \mathsf{Reader}_{+(\alpha\mathsf{DataUser}\setminus\alpha\mathsf{Reader})} \parallel det(\mathsf{DataUser})$$

We encode the elements of this as:

[3] Of course, in this case our tool will loop forever, but this just moves the failure up: If we terminate and then try to run FDR, FDR will run forever (or until it runs out of memory).

```
DataUserplus = DataUser [|{}|] STOP
Readerplus = Reader[|{set}|] STOP
detDataUser = ((get ->DataUser) [] ((set ->DataUser) [] TICK))
```

The latter two processes are combined to form `ReaderplusDet`:

```
ReaderplusDet = Readerplus [|{set,get,tick}|] detDataUser
```

The refinement test can then be applied in FDR by giving the command:

```
Check "DataUserplus" "ReaderplusDet";
```

A similar procedure is followed for the other tests based on refinement. In particular, to carry out the deadlock-freedom based tests we compare the process against the most non-deterministic deadlock-free process with the same alphabet as the process, as we described above.

7.2.2.5 Feedback from FDR

Given two processes, FDR performs the refinement check and provides one of two responses. First, it may verify the refinement, in which case it indicates "`Refinement check succeeded`." This is what happens in our simple example above. Second, it may locate a trace after which there is a failure of the "refining process" that is not a failure of the "specification process," indicating that the refinement test fails. In this case, it informs the user of that trace and of the failure of the "refining" process, thus providing a counterexample to the refinement property. This is how we generated our counterexamples for the HLA case study in chapter 6.

As a simple example, consider a slight variation on the roles DataUser and Reader:

Role DataProvide = get $\rightarrow$DataProvide $\square$ set $\rightarrow$DataProvide $\square$ §
Port ReadOnly = get $\rightarrow$ReadOnly $\square$ §

If we attempt a port-role compatibility check with these definitions, we discover that ReadOnly is not compatible with DataProvide. FDR returns the output below:

```
ReadOnlyplusDet
Interface={|{|get,set,tick|}|}
has behaviour
After <> refuses {|{|set,tick|}|}
```

This output indicates that after the empty trace ($\langle\rangle$) the process `ReadOnlyplusDet` has as one of its refusals $\{$set$,\surd\}$, but the process `DataProvideplus` does not. That is, the trace $\langle\rangle$ is a counterexample to the refinement test. This counterexample leads us to the reason for the incompatibility: DataProvide requires that the event set be handled at any time, but ReadOnly assumes that only get will occur. Thus, ReadOnly is incompatible and we must find a different port to fill this role.

7.2.3 Other Automated Tests

Several of the compatibility and consistency checks are not directly expressed as refinement tests. We have automated four of these.

- "Single initiator" (p. 74) requires that every event be treated as initiated by exactly one process. This can be determined directly from the alphabet information as calculated above.
- "Parameter substitution" (p. 76) requires that parameters supplied to a type result in a consistent non-parameterized type. This is checked by substituting the parameters into the type and then performing the other checks.
- "Range check" (p. 76) requires that any value supplied to a range parameter ($min..max$) fall within the specified range. This is easily determined.
- "Attachment completeness" (p. 79) requires that any un-attached port or role be compatible with $\S$. This is found by applying the port-role compatibility test as described above.

7.2.4 Tests Not Automated

Three of the basic WRIGHT checks are not currently automated. The first is "initiator-commits" (p. 75). Recall that a process is "initiator-commits" if, whenever it can engage in an initiated event, it refuses all other events.

The difficulty with automating this test is that it is not easily expressible as a test of refinement, because it requires the *inclusion* of a particular set of failures, rather than their *exclusion*. Thus, there is no process P' such that P obeys initiator commits whenever P' $\sqsubseteq$ P.

In order to check initiator-commits, we need to check every state of the process and determine that the failures match a certain pattern: whenever there are trace-extensions for an initiated event and for another event, then the initiated event is chosen non-deterministically. This is a straightforward exhaustive checking problem, in that it involves generating all of the states of an automaton and checking that they obey a particular, well-defined property. Thus, there are known techniques that can be used to implement this check (by looking at every state of the process). In particular, we could use the internal state graph calculated by FDR, and simply traverse it, checking every state. Implementation of this check is planned for the near future.

The other tests we have not implemented are to determine whether a configuration obeys its declared style constraints (p. 78) and whether a style has legal configurations (p. 78). As we discuss later (chapter 9), for the general form of constraints that we have permitted, and in particular for semantic constraints (on the behavior of components, connectors, and the system as a whole), determining style conformance is undecidable. It is conceivable that an automatic checker could be constructed for those cases where a system is finite-state. However, an exhaustive check at the global level, as required by some style constraints, is likely to be prevented by the practical limitations described earlier.

An alternative to automating these checks in their full generality is to automate a subset of the language. One such subset is to cover topological constraints and use name-matching on type constraints (*i.e.*, expressions of the form "$Type(i) = t$" where i is an instance variable and t is the name of a declared type). This would provide quite a bit of help to architects without being unduly complicated. All that would be needed to check a configuration's conformance to constraints is a boolean expression evaluator.[4] Developing this restricted checker is also short-term future work. To check style consistency for this restricted subset, we could apply a predicate based exhaustive checker such as Nitpick [JD96].

7.3 Providing General Analysis Automation

Automated support for analysis of WRIGHT specifications is not limited to the standard checks. FDR provides a much more general capability for analysis of CSP processes, and users of WRIGHT can take advantage of this. Our automated tools, in addition to applying the checks, generate the CSP process that is the meaning of an architectural type or configuration as defined in chapter 3. These processes can be used by the architect to have FDR perform any analysis that can be expressed as a refinement check. For example, suppose we wanted to ensure that the RTI connector described in the HLA case study requires that it be destroyed before the system terminates. We can check this in a way similar to how we checked deadlock-freedom above. In FDR's syntax, we describe a process that always destroys before terminating:

```
DoesDestroy = (|~| i :  NFEDS @ Fed.i.destroyFedExecution
                                -> TICK)
```

We then hide all of the events except destroyFedExecution and $\surd$ in the RTI process:

```
RTIDestroy = RTI \ setminus(setminus(ALPHARTI,
                    {tick}),
                    { i :  NFEDS @ Fed.i.destroyFedExecution})
```

Then we can check whether the RTI is guaranteed to be destroyed:

```
Check "DoesDestroy" "RTIDestroy";
```

This fails, of course, because of the deadlocking problems described in chapter 6. However, we only wanted to ensure that the RTI *requires* that destruction be handled correctly. The deadlock problem can be avoided by ignoring refusals and looking only at matching traces. This is checked by using `CheckTrace` instead of `Check`. In this case, a trace-refinement of `DoesDestroy` ensures that *if* the system terminates without deadlocking (engages in $\surd$), *then* destroyFedExecution must have occured previously. The RTI passes this test, indicating that we have correctly specified the required use of destroyFedExecution.

As we discussed above, there are practical limits on the ability of FDR to check properties on configurations, due to the complexity of the process corresponding to the full system. However, this technique seems promising for component and connector types, which are local and therefore simpler. Also, results on a type can be applied across all configurations that use that type, further increasing the leverage of the analysis.

[4] Of course, there are efficiency issues surrounding the $\forall$ and $\exists$ operators that need to be addressed. There are no difficult *semantic* issues, however.

Chapter 8

Data Models for HLA

8.1 Introduction

In the previous chapters we have seen how WRIGHT's model of architectural structure and abstract behavior provides an expressive, practical notation for the description of software architecture configurations and styles. By providing a model of abstract behavior based on CSP, WRIGHT supports the specification and analysis of properties such as interaction and control. We have also seen how we can use commercial tools to automatically check WRIGHT specifications for consistency and completeness based on those CSP specifications.

There are properties and issues for architecture that are not naturally captured in CSP, however. For example, complex state invariants are not easily expressed in terms of events. While it is possible to model state through a combination of subscripts on processes and the "?" and "!" event operators, these are not explicitly part of the CSP semantics. Each "?" operator, for example, introduces a family of separate events, rather than a single event with parameters. Similarly, the CSP failures model (described in appendix A) does not include the name of a process or its subscripts as part of its semantics, and so it is difficult to use that information as part of an analysis.

This problem of focussing on some properties at the expense of others is fundamental to any notation. The essence of abstraction is to promote some issues and properties by ignoring others. That is, description and analysis of a given property is simplified by using a notation that exposes that property as fundamental and ignores other properties. In the HLA case study (chapter 6), for example, we *deliberately* ignored state information: this allowed us to simplify our description of control and made our analysis tractable.

At the same time, any development activity will need to consider *all* of the critical issues that affect the success or failure of a system. Thus, an architect will want to consider multiple views of an architecture, each view exposing one or more issues while suppressing others, thus supporting the abstract consideration of each issue.

As we discuss in chapter 10, the general problem of combining formalisms to provide rich, multi-viewed descriptions of software architecture is a significant research challenge, and one that is not a primary focus of this thesis. Approaches that have been taken include defining mappings from multiple domains into a single, "least common denominator" model [ZJ93], using a single paradigm but multiple views [Jac95a], and developing models that combine multiple phenomena for specific domains [AD94] or for specific combinations of phenomena [Hav90].

Another possible approach is to use different notations for different abstractions, but to structure the specifications so that simple correspondences (*i.e.*, mappings) between the two can be defined. For example, we can define a state model for an architecture but structure it to match a WRIGHT model by having every state-change correspond to a specific event. These are combined into sequences of state changes corresponding to each WRIGHT trace.

In this chapter we explore this approach. We will illustrate the approach using the HLA style (described in chapter 6), and show how one state oriented property, expressed in Z [Spi92] can be combined with the WRIGHT specification of the style.

By using the WRIGHT specification of HLA as a framework, we take advantage both of that specification's architectural structure and its model of a system's dynamic behavior. This allows us to simplify the Z model (by not repeating the structure and control models contained in the WRIGHT specification), keep the level of abstraction high (by directly expressing the properties of interest) and at the same time to relate the model to both the architectural structure and the behavior of the systems.

We begin our description by characterizing an abstract system state that captures the attribute ownership property. Then, we will describe some state-consistency properties as desired invariants over the state model. Note that these are *desired* properties. The idea of the model is to discover whether the protocols described in the WRIGHT specification, together with state-specific preconditions described in the Z model, are adequate to ensure that these invariants hold.

After modelling the system state and describing the desired invariants we will model how the state can be changed by federates and the RTI. Each state change corresponds to a single WRIGHT event, providing a basis for combining the two models. Each message will be described separately. This will both simplify the specification and provide direct traceability to the informal specification, because each (state-based) message transition can be compared directly to the pre- and post-condition descriptions of that document.

The fourth and final step of creating the model will be to combine it with the WRIGHT model of the HLA, developing a dynamic model where the effect of messages are combined in sequences, matching the control behavior described in WRIGHT. Although we will not carry out any specific formal proofs using the developed model, we will conclude with a discussion of how that kind of analysis might be carried out, using a combination of the automated tools available both for WRIGHT and for a subset of Z (*i.e.*, Nitpick [JD96]) and inductive techniques similar to those applied in the JC case study (chapter 5).

8.2 State Model of Attribute Ownership

In an HLA federation, a simulation state is modelled as a collection of objects, each of which has a collection of attributes with particular values. Computation is divided among federates by assigning each of the federates a subset of the attributes to calculate. A federate that is calculating an attribute's value and providing that value to other federates is said to be "publishing" that attribute. A federate that needs to know the value of an attribute "subscribes" to the attribute, indicating that any updates should be passed to that federate. In addition, a federate can own an object, in which case it has the right to delete that object.

When an object is deleted, all federates are expected to stop publishing attributes for that object, and all subscribers are notified.

8.2.1 Model of State

We begin by modeling the collection of objects that are part of a simulation: A collection of objects, each of which have some named attributes.

$$[OBJECT, ATTRIBUTE]$$

$$\begin{array}{|l}
\hline
\textit{ObjectCollection} \\
\hline
Objects : \mathbb{P}\ OBJECT \\
ObjectAttrs : OBJECT \leftrightarrow ATTRIBUTE \\
\hline
\mathrm{dom}\ ObjectAttrs = Objects \\
\hline
\end{array}$$

Next we model the publication and subscription of attributes and object ownership.

$$[FEDERATE]$$

$$\begin{array}{|l}
\hline
\textit{SimulationState} \\
\hline
ObjectCollection \\
Federates : \mathbb{P}\ FEDERATE \\
Publishing : FEDERATE \leftrightarrow (OBJECT \times ATTRIBUTE) \\
Subscribing : FEDERATE \leftrightarrow (OBJECT \times ATTRIBUTE) \\
DeletePriv : FEDERATE \leftrightarrow OBJECT \\
\hline
\mathrm{dom}\ Publishing \subseteq Federates \\
\mathrm{dom}\ Subscribing \subseteq Federates \\
\mathrm{dom}\ DeletePriv \subseteq Federates \\
\hline
\end{array}$$

The $SimulationState$ schema indicates that at any given point in a simulation execution, there is a collection of objects ($ObjectCollection$) and a collection of federates ($Federates$). Each federate publishes some collection of object attributes ($Publishing$), subscribes to some collection of object attributes ($Subscribing$), and is permitted to delete some collection of objects ($DeletePriv$).

There are two things worth noting about this definition: (1) it is a direct recasting of the informal description, and (2) it does not contain many of the consistency constraints one might expect. For example, it does not require that there be at most one federate publishing a given object attribute.

This last point is important: We want to provide a model in which things can go wrong, and then show that the architecture and its operations ensure that those things do not go wrong.

8.2.2 Desired Invariants

In order to locate potential inconsistencies in a system, we must define what we mean by a consistent state. Here are a few examples of consistency constraints:

No federate subscribes to an object that does not exist.

$$\begin{array}{|l} \hline \textit{GoodSubscribes} \\ \hline SimulationState \\ \hline \operatorname{ran} Subscribing \subseteq ObjectAttrs \\ \hline \end{array}$$

Every object attribute is published by exactly one federate, and no federate publishes an attribute that does not exist.

$$\begin{array}{|l} \hline \textit{GoodPublishes} \\ \hline SimulationState \\ \hline Publishing^{\sim} \in (ObjectAttrs \longrightarrow Federates) \\ \hline \end{array}$$

Every existing object is deletable by exactly one federate, and no federate may delete any non-existent object.

$$\begin{array}{|l} \hline \textit{GoodDeletes} \\ \hline SimulationState \\ \hline DeletePriv^{\sim} \in (Objects \longrightarrow Federates) \\ \hline \end{array}$$

We combine these consistency invariants into a single schema for convenience:

$$GoodSimulation \mathrel{\widehat{=}} GoodSubscribes \wedge GoodPublishes \wedge GoodDeletes$$

8.2.3 Operations

Next we must define how a simulation state can change. Each message defined in the HLA interface specification has the potential to change the system state (as defined by $SimulationState$). Thus, we can define a state change for each message, indicating what its effect on the state is. Each message has an $Operation$ schema associated with it:

$$\begin{array}{|l} \hline \textit{Operation} \\ \hline \Delta SimulationState \\ fed? : FEDERATE \\ \hline \end{array}$$

Here are some examples of how we might define the effect of messages:[1]

[1] Although we will only show a few key examples, every message defined in the HLA style would have an *Operation* schema associated with it. Most would include $\Xi SimulationState$ to indicate that they do not change the attribute ownership state.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CreateFedExecution}} \\
\hline
Operation \\
\hline
Objects' = \emptyset \\
Federates' = \emptyset \\
\hline
\end{array}
$$

Until the createFedExecution message is sent by some federate, the simulation does not logically exist. Therefore, this message defines the initial state. There are initially no objects or federates.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{JoinFedExecution}} \\
\hline
Operation \\
\hline
fed? \notin Federates \\
Federates' = Federates \cup \{fed?\} \\
ObjectAttrs = ObjectAttrs' \\
Publishing = Publishing' \\
Subscribing = Subscribing' \\
DeletePriv = DeletePriv' \\
\hline
\end{array}
$$

When a federate joins the execution, it becomes part of the state, but does not otherwise affect attribute ownership.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RequestAttrOwnDivestiture}} \\
\hline
Operation \\
\Xi SimulationState \\
obj? : OBJECT \\
attrs? : \mathbb{P}\ ATTRIBUTE \\
\hline
\{fed?\} \times (\{obj?\} \times attrs?) \subseteq Publishing \\
\hline
\end{array}
$$

When a federate sends the message reqAttrOwnDivestiture, it provides an object and a collection of attributes that it wishes to divest. The post-conditions described in the HLA interface specification state that the federate should not stop publishing the object. It will be a proof obligation to show that the protocol does indeed eventually result in an attrOwnDivestNotify message, indicating that the federate may stop publishing:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AttrOwnDivestNotify}} \\
\hline
Operation \\
obj? : OBJECT \\
attrs? : \mathbb{P}\ ATTRIBUTE \\
\hline
\{fed?\} \times (\{obj?\} \times attrs?) \subseteq Publishing \\
Publishing' = Publishing \setminus (\{fed?\} \times (\{obj?\} \times attrs?)) \\
Subscribing = Subscribing' \land DeletePriv = DeletePriv' \\
ObjectAttrs = ObjectAttrs' \\
\hline
\end{array}
$$

Join Federation Execution

Federate Initiated

..informal description text..

Supplied Parameters
Federate Name
Federate ID
a Federation execution name
Connection parameters, if required
Optional federation time when connection will occur

Returned Parameters
Current value of federation time
Federation state information (to be defined later).
Federation binding ID

Pre-conditions
The named federation execution exists.
The federate ID is not joined.

Post-conditions
The federate is a member of the named federation execution.

Exceptions
.. preconditions violated ..

Figure 8.1: A sample of the informal simulation definition.

Notice how in the schema $AttrOwnDivestNotify$, the condition $GoodPublishes$ can not be true in both its pre- and post-states. Thus, part of our analysis should consider whether the RTI Glue ensures that divestitures will always be paired with acquisitions:

$$
\begin{array}{|l}
\hline
\textit{AttrOwnAcquisitionNotify} \\
\hline
Operation \\
obj? : OBJECT \\
attrs? : \mathbb{P}\ ATTRIBUTE \\
\hline
Publishing' = Publishing \cup (\{fed?\} \times (\{obj?\} \times attrs?)) \\
Subscribing = Subscribing' \wedge DeletePriv = DeletePriv' \\
ObjectAttrs = ObjectAttrs' \\
\hline
\end{array}
$$

Each of these schemas are derived from the pre- and post-conditions of the message specification in the official document. For example, $JoinFedExecution$ is defined as in figure 8.1 [DMSO95, p. 16]. Note how the informally specified pre- and post-conditions match the formal conditions in our Z model.

8.3 Combining State Model with Behavior Model

So far we have described a model of the system state that exposes a particular issue of interest: object attribute ownership. We have shown what the state elements are and how that state is changed by individual messages. We also noted that we may need to reason about sequences of messages in order to demonstrate that the desired invariants hold. Thus, we must consider the dynamic behavior of the system.

We model a system execution as a sequence of system state changes, each one corresponding to the effect of some message:

$$\begin{aligned} LegalOperation \mathrel{\widehat{=}} {} & CreateFedExecution \lor JoinFedExecution \\ & \lor RequestAttrOwnDivestiture \lor AttrOwnDivestNotify \\ & \lor AttrOwnAcquisitionNotify \end{aligned}$$

$$\begin{array}{l} SimulationTrace : \mathbb{P}(\mathrm{seq}\ Operation) \\ \hline \forall s : SimulationTrace;\ i : \mathbb{N}_1 \mid i \in 1 \mathrel{..} \# s \bullet \\ \quad i < \# s \Rightarrow \\ \qquad (\exists\, \Xi SimulationState \bullet \\ \qquad\quad (\exists\, SimulationState; fed? : FEDERATE \bullet s(i) = \theta Operation) \\ \qquad\quad \land (\exists\, SimulationState'; fed? : FEDERATE \\ \qquad\qquad \bullet s(i+1) = \theta Operation)) \\ \quad \land (\exists\, LegalOperation \bullet s(i) = \theta Operation) \end{array}$$

A $SimulationTrace$ is a sequence of $LegalOperation$s such that the post-state of each operation is the pre-state of the operation following it.[2]

We could take the set of all possible $SimulationTrace$s to be our architectural model, and attempt to demonstrate consistency. This has one important drawback: The message descriptions above do not capture enough information about what constitutes a legal execution. Many potential traces permitted above are not actually legal. For example, nothing in the above definition requires that a simulation execution begin with a $CreateFedExecution$. This means that there will be many inconsistent traces in the set $SimulationTrace$ that are prevented by the standard, and therefore not of concern.

One possibility would be to add constraints to $SimulationTrace$ to try to capture these other restrictions on traces. However, this would require considerable effort. Essentially, we would need to construct trace constraints equivalent to the WRIGHT model described in chapter 6. Instead, we can use the WRIGHT specification itself to provide this information. We do this by defining a mapping between the Z schemas and events in the WRIGHT model. Here we assume that there are functions transforming the $(alphabet, failures, divergences)$ model of CSP so that it can be accessed in Z. To access this model, we define given types for processes and events, and functions α and $traces$. These functions should be interpreted as corresponding to the definitions in appendix A.

[2]The careful reader will notice that the *obj*? and *attrs*? parameters are hidden in a *SimulationTrace*. We have done this because there is currently no part of the specification that shows how those parameters must correspond across sequences of operations. A fuller model would, of course, have to take them into account.

$$[PROCESS, EVENT]$$

$$\begin{array}{|l} \alpha : PROCESS \longrightarrow \mathbb{P}\ EVENT \\ traces : PROCESS \longrightarrow \mathbb{P}(\mathrm{seq}\ EVENT) \end{array}$$

In addition, for the HLA style as defined in WRIGHT, we define a process RTI that is the meaning of the connector type RTI in the WRIGHT definition, and HLA, which is the set of all configuration processes that obey the HLA style.

$$\begin{array}{|l} RTI : PROCESS \\ HLA : \mathbb{P}\ PROCESS \end{array}$$

Given these definitions, we can define a correspondence between $Operation$s in the Z and events in the WRIGHT.

$$\begin{array}{|l} WrightTrace : \mathbb{P}(\mathrm{seq}\ Operation) \\ WrightStep : Operation \leftrightarrow \alpha RTI \\ \hline WrightStep = \{d : CreateFedExecution \bullet (d, \mathsf{Fed}_{d.fed?}.\mathsf{createFedExecution})\} \\ \qquad\qquad \cup\{d : JoinFedExecution \bullet (d, \mathsf{Fed}_{d.fed?}.\mathsf{joinFedExecution})\} \\ \qquad\qquad \cup\{d : Operation \mid (\exists\ RequestAttrOwnDivestiture \bullet \theta Operation = d) \\ \qquad\qquad\qquad\qquad \bullet (d, \mathsf{Fed}_{d.fed?}.\mathsf{requestAttrOwnDivestiture})\} \\ \qquad\qquad \cup\{d : Operation \mid (\exists\ AttrOwnDivestNotify \bullet \theta Operation = d) \\ \qquad\qquad\qquad\qquad \bullet (d, \mathsf{Fed}_{d.fed?}.\mathsf{requestAttrOwnDivestiture})\} \\ \qquad\qquad \cup\{d : Operation \mid (\exists\ AttrOwnAcquisitionNotify \bullet \theta Operation = d) \\ \qquad\qquad\qquad\qquad \bullet (d, \mathsf{Fed}_{d.fed?}.\mathsf{requestAttrOwnDivestiture})\} \\ WrightTrace = \{System : HLA; s : \mathrm{seq}\ Operation; t : \mathrm{seq}\ EVENT \\ \qquad \mid t \in traces(System) \\ \qquad\qquad \wedge \#s = \#t \wedge (\forall\ i : 1\ldots\#s \bullet (s(i), t(i)) \in WrightStep) \\ \qquad \bullet s\} \end{array}$$

Each element of $WrightStep$ defines a correspondence between a state change ($Operation$) and a WRIGHT event (αRTI). The definition of $WrightStep$ is constructed by providing one set definition for each event in the set αRTI, of the form $\{d : OperationSchema \bullet (d, \mathsf{Fed}_{d.fed?}.e)\}$. If the WRIGHT event abstracts away from some parameters of the operation, then those are hidden before the pair is constructed.

Once the set of $WrightStep$s is defined, they are built up into a trace by using the behavior of a configuration in the HLA style as defined by WRIGHT ($System$).

We can now combine the information of the two models by ensuring that any trace we consider is permitted by both:

$$\begin{array}{|l} BothLegalTrace : \mathbb{P}(\mathrm{seq}\ Operation) \\ \hline BothLegalTrace = SimulationTrace \cap WrightTrace \end{array}$$

8.4 Analysis of the Combined Specification

By creating a formal specification that describes a particular property (attribute ownership, in our example), and defining correspondences with the WRIGHT specification, we have created a model in which we can reason about whether and how systems in the HLA style deal with the property. Above, we defined a set of desired invariants in the schema $GoodSimulation$ and a set of legal traces through the trace set $BothLegalTrace$. We can relate these as defining a $GoodTrace$ to be one that establishes the invariant, and $BadTrace$ as one that results in the violation of that invariant:

$$
\begin{array}{|l}
GoodTrace : \mathbb{P}\ BothLegalTrace \\
BadTrace : \mathbb{P}\ BothLegalTrace \\
\hline
GoodTrace = \{ t : BothLegalTrace \mid \exists\ Operation \bullet t(\#t) = \theta Operation \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ GoodSimulation' \} \\
BadTrace = \{ t : BothLegalTrace \mid \exists\ Operation \bullet t(\#t) = \theta Operation \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \neg\ GoodSimulation' \}
\end{array}
$$

One possible analysis is to consider whether all legal executions of the the architecture maintain the desired invariants at all times:

$$BothLegalTrace = GoodTrace$$

As we discussed above, the messages do not maintain ownership atomically, so this hypothesis is not what we want. We want instead to reason about whether the protocol ensures that *eventually* consistency is re-established. One possible expression of this is as follows:

$$
\begin{array}{l}
\forall\ bad : BadTrace \bullet \\
\qquad \exists\ cont : \mathrm{seq}\ Operation \bullet bad \frown cont \in GoodTrace
\end{array}
$$

This hypothesis states that whenever the computation breaks the consistency invariants, it is always possible to re-establish the consistency.

Here is an arbitrary, stronger hypothesis. It states that consistency is always re-established in exactly three steps.

$$
\begin{array}{l}
\forall\ bad : BadTrace;\ cont : \mathrm{seq}\ Operation \\
\qquad \mid bad \frown cont \in BothLegalTrace \wedge \#cont = 3 \\
\qquad \bullet\ bad \frown cont \in GoodTrace
\end{array}
$$

In order to reason about the truth or falsity of these hypotheses, we can consider both the possible traces of the WRIGHT specification and the pre- and post-conditions of the Z specification. One way of considering these is through the use of automated property checkers such as FDR, discussed in the previous chapter. For a subset of Z, there is a checker called Nitpick. Nitpick considers relational expressions (such as Z schemas) and searches for values of variables such that the expressions are false.

While one can not directly combine FDR and Nitpick, they can be used to generate intermediate results in a combined proof. For example, we could use the Z specification and Nitpick to calculate two sets of messages: those that are capable of violating the consistency invariant, and those that will re-establish the invariants. Then, we can use FDR and the WRIGHT specification to show that whenever any event in the first set occurs it will necessarily be followed by one in the second set.

8.5 Discussion

In this chapter we have explored a particular technique for combining alternative semantic models with WRIGHT specifications. We have shown how a Z specification of an abstract system state can be combined with a WRIGHT specification. To do this, we structured the Z specification in a particular way: every possible state change (Δ schema) corresponds to a single event in the WRIGHT model. This correspondence permits us to map the state model into a system execution by constructing sequences of state changes that correspond to a trace of the WRIGHT event model.

This technique has several advantages: It allows us to keep the WRIGHT specification simple, by retaining pure event oriented abstractions. It simplifies the state model by allowing us to omit sequence-of-event constructs. It means that we can reason directly in the semantic domain of each separate notation (Z and CSP) and use tools such as Nitpick and FDR. Through its correspondence to the WRIGHT style specification, the Z model also provides a model of a family of systems.

This last point is important: Because we have set up the model to correspond to a WRIGHT specification *of a style*, the Z model also describes a style. That is, the WRIGHT structures that extend an analysis from a style to a configuration also apply to the state model that we have developed. Any configuration that is shown to be in the HLA style can take advantage of results of analyzing both the WRIGHT specification and the Z state-model.

Chapter 9

Discussion and Evaluation

In this chapter we review the requirements for a practical basis for architecture description and analysis, and consider how WRIGHT fulfills them. In order to do so, we consider a number of design decisions, and ask whether they might have been made differently, and what the consequences of that would be. First, recall the requirements:

- *Description of architecture configurations:* The ultimate goal of architectural description is to provide an effective means of describing the architecture of a software system. It must be possible to define the components that will occur in a system and the interactions between them.

- *Description of architectural styles:* Beyond describing single systems, architects should be able to delineate the members of a *family of systems*. These architectural styles provide a means of exploiting commonalities between systems and for leveraging analysis and implementation efforts. Additionally, the two levels of description should be able to be combined — given a configuration, we want to know if it is a member of a given style.

- *Analysis of properties of interest:* An important goal of the descriptive activity is to use the description to understand properties of the system that has been or will be constructed. The architect must be able to use a description to analyze the system or style to determine whether it satisfies its requirements. Note that it is important to analyze both individual configurations and architectural styles. Ultimately, it is the properties of individual systems that are important, but analysis of styles provides important leverage by applying to more than one architectural configuration.

- *Application to practical problems on real systems:* While *some* analytic leverage is arguably an improvement over no ability to analyze a description, a notation is not generally practical if it can only be applied in tightly constrained circumstances or only on small systems. A practical notation must address this by scaling to apply to complex, real-world problems.

We now consider each of these in turn, and show how WRIGHT fulfills them. By showing that WRIGHT satisfies these requirements, we will have shown that a formally based architecture description language can provide a practical means of describing and analyzing software architectures and styles.

9.1 Description of Configurations

In this section, we discuss three issues relating to WRIGHT's description of architectural configurations. We will discuss WRIGHT's treatment of connectors, the question of whether CSP might have been used directly to specify architectures, and the problem of dynamic reconfiguration of configurations.

9.1.1 Connectors

Probably the most important aspect of WRIGHT is its treatment of connectors. By treating connectors as explicit, abstract specifications of interaction patterns between components, WRIGHT supports the description and analysis of the behavior of architectural configurations. In this section, we address three questions relating to WRIGHT's treatment of connectors: First, whether explicit connectors are necessary; second, whether WRIGHT's use of general glue specifications could be simplified; third, whether and how WRIGHT could usefully be extended to model synchronous interactions through its connectors.

9.1.1.1 Why have them?

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

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

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

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

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

We believe that there are many cases, such as the RTI connector in chapter 6, where the relation between the events of different components' ports is complex enough to require the kinds of descriptive power that WRIGHT offers through its general **Glue** mechanism. It is precisely for these non-trivial connectors that the formal approach is most valuable. It is therefore appropriate to include the ability to describe these general connectors.

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

Another important benefit from the approach that we have taken is its support for architectural styles. Most styles are constructed around a small set of interaction types, such as rpc, event broadcast, *etc*. By separating the relatively fixed specifications of connectors from the relatively variable specifications of components, WRIGHT more easily supports this kind of structure in a style. Moreover, the definition of port-role compatibility is directly responsive to connector reuse, since it permits many different kinds of component ports to be used with the same kind of connector role.

One particular example of a simple, but useful, application of explicitly described interactions is the resolution of name-mismatch problems. When constructing a system from pre-existing components, such as libraries, an architect is often faced with the problem that two components that ought to be usable together use different names for the various routines that are used to communicate. In order to use these components together, it becomes necessary to modify one or both of the components to make the names match. Then, there are multiple versions of that component, the original and the newly renamed versions, and no explicit mapping from one to the other. By making the correspondence between events in different components explicit and separate from the components' definitions, WRIGHT connectors provide a means of describing the correspondence between the different versions explicitly, thus supporting the effective reuse and maintenance of the components in question. There is still only one version of the component specification, and the necessary modification for reuse is described separately as a connector for a specific reuse context.

9.1.1.2 Is Glue too cumbersome?

While we have argued above that it is appropriate to include the ability to describe connector interactions generally, as we have in WRIGHT, many connectors could be expressed by a simple mapping of one component's events to another's, often by simple name matching. In these cases, the **Role** definitions by themselves completely characterize the protocol. So why bother with the **Glue** at all?

For example, a procedure call connector has two roles, a definer and a caller, each of which specifies the request-then-result pattern of interaction:

Connector ProcedureCall

$$
\begin{array}{ll}
\textbf{Role } \mathsf{Definer} = & \mathsf{request} \rightarrow \overline{\mathsf{result}} \rightarrow \mathsf{Definer} \\
& \square\ \S \\
\textbf{Role } \mathsf{Caller} = & \overline{\mathsf{request}} \rightarrow \mathsf{result} \rightarrow \mathsf{Caller} \\
& \sqcap\ \S \\
\textbf{Glue} = & \mathsf{Definer.request} \rightarrow \overline{\mathsf{Caller.request}} \rightarrow \textbf{Glue} \\
& \square\ \mathsf{Caller.result} \rightarrow \overline{\mathsf{Definer.result}} \rightarrow \textbf{Glue} \\
& \square\ \S
\end{array}
$$

As we can see, the **Glue** simply shuttles the request event from caller to definer and the result from definer to caller. One might argue that it would simplify WRIGHT to include structures for omitting or simplifying the **Glue** description in connectors where the correct matching of events is “obvious.”

We have chosen not to include this kind of syntactic structure as primitives in WRIGHT for two reasons: (1) These additions do not add analytic power or simplify reasoning in WRIGHT, and (2) use of overly simple connector definitions can weaken the leverage gained by using WRIGHT.

To elaborate, these simple cases are already expressible as in our example above, and so clearly a special syntax for them would not add analytic power to WRIGHT. Further, adding special semantic interpretations of an omitted glue would not simplify the underlying semantic model or methods for reasoning about these connectors. The repetition of a protocol in multiple places in a connector description (in **Role**s and **Glue**) does not result in additional constructs at the semantic level in WRIGHT’s model. The **Role** protocols provide descriptions that do not appear in the interpretation of a configuration. These specifications need to occur anyway (as we argued in chapter 3) to ensure the independence of connectors from their eventual context of use. The use of a default **Glue** mechanism simply means that the underlying configuration model contains constructs that are not explicitly spelled out in the description. The process described by the explicit **Glue** would still need be present in the underlying semantics, to maintain the correspondence of events between different component instances (which have different names to ensure the independence of component type descriptions).

Further, the WRIGHT style description facilities can be used to define simple syntaxes for basic correspondences. For example, consider the basic one-to-one event pairings that are typical in simple interactions such as rpc or client-server connectors. In this case, when all of the event names match up and there is no overlap in initiated events, WRIGHT’s existing definition facilities can be used to describe the following utility process:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{Process } \mathsf{NameMatch}_{\mathsf{Role}_1,\mathsf{Role}_2} =} \\
& (\square\ e : \alpha_i \mathsf{Role}_1 \bullet\ \mathsf{Role}_1.e \rightarrow \overline{\mathsf{Role}_2.e}) \\
\square & (\square\ e : \alpha_i \mathsf{Role}_2 \bullet\ \mathsf{Role}_2.e \rightarrow \overline{\mathsf{Role}_1.e}) \\
\square & \S
\end{array}
$$

Given two roles, NameMatch constructs a process that observes all of the initiated events of both roles, transmitting them from one role to another. NameMatch can then be used to define the **Glue** for a connector:

Connector ProcedureCall
  Role Definer = request$\rightarrow$ $\overline{\text{result}}$$\rightarrow$Definer
$\square\ \S$
  Role Caller = $\overline{\text{request}}$$\rightarrow$ result$\rightarrow$Caller
$\sqcap\ \S$
  Glue = NameMatch$_{\text{Definer,Caller}}$

NameMatch can be used in any situation to describe a simple name-correspondence between roles, and so it is not necessary to include this kind of construction as a primitive in WRIGHT.

The second reason we have not included these syntactic simplifications in WRIGHT's core language is that the benefits of WRIGHT arise from the precise definition of the semantics of architectural entities, and overly simple connector definitions dilute the usefulness of WRIGHT's checks.

A very simple, common kind of connector, for example, is a definition/use connector, where one component defines a collection of calls that it accepts, and another component uses them, by requesting that they occur and receiving a result. These roles could be constructed, for example, using these utility definitions:

DefineSet$_E$ = ($\square\ e : E \bullet e$.request $\rightarrow\overline{e\text{.reply}}$ $\rightarrow$DefineSet$_E$) $\square\ \S$
UseSet$_E$ = ($\sqcap\ e : E \bullet \overline{e\text{.request}}$ $\rightarrow e$.reply $\rightarrow$UseSet$_E$) $\sqcap\ \S$

The definition/use connector might then be:

Connector DefUse($E : \mathbb{P}\ \Sigma$)
  Role Definer = DefineSet$_E$
  Role User = UseSet$_E$
  Glue = NameMatch$_{\text{Definer,User}}$

In the DefUse connector, one participant, Definer, provides a set of services, but does not restrict how they are to be used. Another participant, the User, will request some or all of these services, in some order, without any details about the protocol except that a reply is expected. The **Glue** ensures that service requests and replies are properly communicated between the parties.

This is, in fact, exactly the semantics that are attributed to connectors in many ADLs and module interconnection languages. Further, the DefUse connector type satisfies the consistency checks defined for WRIGHT connectors. (NameMatch is always prepared to accept an event, and is therefore deadlock-free, and the construction of matching initiations/observations ensures initiator commits and single initiation.) But if this connector is used in a situation where there are, in fact, constraints on the ordering of requests, as is typical, the problems that the WRIGHT checks are meant to prevent will still occur. That is, it is still perfectly possible for the component that fills the Definer role to expect the requests to occur in a particular order, and thus for the configuration to fail in execution when the User doesn't live up to that expectation. Also, the connector says essentially nothing about what will happen, so any analysis of the properties of a configuration that use this connector type will have to be delayed until it is instantiated, *i.e.*, at a global rather than local level of the architecture.

Thus, we claim that WRIGHT has struck the appropriate balance between descriptive power and simplicity regarding these kinds of connectors. If a simple, name-matching connector is in fact the right connector for a given architecture, then it can be described in WRIGHT either directly at the cost of some redundancy, or through the use of shorthand process definitions such as those described above, without any additional cost. These common cases could easily be collected together in a style, so that the architect could use the vocabulary directly in a WRIGHT configuration definition. On the other hand, these cases do not inappropriately dominate the syntax and semantics of WRIGHT, and so the architect is encouraged to use the full richness of detail and analysis that is available.[1]

9.1.1.3 Is Glue too weak?

While many kinds of connectors can be specified using WRIGHT, there is an important semantic property of all WRIGHT connectors: They are asynchronous. This is because all components have disjoint event alphabets. The connector synchronizes with each component, but there is no "atomic" operation involving two components.

Asynchrony is currently an intrinsic part of the WRIGHT semantics. From the point of view of describing certain behaviors, where there are atomic, shared control abstractions available to the underlying system (*e.g.*, rendezvous in Ada), this is a drawback. While it would be possible to encode (the implementation of) a synchronous operation directly as a connector, this isn't really suitable, since it lowers the level of abstraction, rather than raising it. That is, Ada provides a synchronization as a single, abstract, concept. By forcing the architect to break that concept out into multiple events (a hand-shake, for example), we reduce the level of abstraction by including details (of Ada's implementation!) in the architectural specification that are not included in the supposedly more concrete description.

In order to alleviate this problem, we could have augmented the syntax and semantics of WRIGHT to include a direct component-component synchronization. This has not been done for a technical reason that will be described below.

The way that a synchronous connector might be handled is as follows. We would change WRIGHT's **Glue** construct to have two parts: First, as currently, a behavior process that makes correspondences between events via ordering (→) and choice (□, ⊓). Second, an additional set of event aliases. These would take events from different roles and declare that they are actually the same event. For example, using this alternate syntax, we might describe a synchronous procedure call:

Connector SynchroProcedureCall
  Role Definer = request→ $\overline{\text{result}}$→Definer
    □ §
  Role Caller = $\overline{\text{request}}$→ result→Caller
    ⊓ §

[1]Of course, no language can force someone to have or use good taste in a description. We merely suggest that WRIGHT supports the architect's efforts to find a good description by making it easy to consider alternative levels of detail and abstraction.

Glue = **RUN**
  aliases Definer.request = Caller.request
    Definer.result = Caller.result

SynchroProcedureCall is similar to the ProcedureCall connector above, but it indicates that the events are actually the *same event*, rather than being transmitted by a **Glue** mechanism. This behavior is typical of the connectors that would be encoded with the NameMatch process described above. In these situations, the intended semantics is usually that there is no mechanism (asynchronous or otherwise) that is associated with the connector, but that the components communicate without any intermediaries.

As an alternative to this synchronous mechanism, in situations where one of the components is strictly in control of each event (as occurs in procedure call, for example), we can approximate synchrony in WRIGHT's current form by using matched events, one initiated and one observed, and having the **Glue** prevent any events between them. Because no event can occur between the two events there is no way to observe the asynchrony within the context of a single interaction.

This approximation has two weaknesses, however. First, at a global (configuration) level, no single **Glue** controls all events, and there can be race conditions where "synchronous" events on two different connectors appear to occur in different orders for different components. Second, mixed-control synchronizations (where both parties must agree for rendezvous to take place) are not accurately described with a simple event pair.

Technical details

In order to add a true synchronous aliasing construct to WRIGHT, it would be necessary to modify the semantics of the language. One possibility is to define a renaming function for each connector instance that has such constructs. The renaming function would rename the events of a component instance's port (*e.g.*, ProcUse.Procedure.request) to the events of another component instance's port (*e.g.*, ProcDef.Definition.request). The renaming function would be applied to the appropriate instance before it is placed in parallel with the rest of the system, at which point it would synchronize with the other component. The apparently simpler scheme of providing a renaming function per connector type will not work because the port's name is not known until instantiation. Thus, interpreting the synchronous construct requires a more complicated model of instantiation than WRIGHT currently has, since instantiation of types and attachments must be handled simultaneously.

The main technical difficulty with this approach, however, has to do with connector self-loops. A self-loop is a connector instance that has one component that is attached to the same connector more than once via different ports and roles.

The problem is that, in CSP, if one were to take a process such as

$$P = a \rightarrow R \mathbin{[]} b \rightarrow Q,$$

and rename two events in the process so that they now have the same name, the result is two events with the same name, not one event. For example, take P, and rename both a and b to c. The resulting process is

$\mathsf{P}' = \mathsf{c} \rightarrow \mathsf{R}' \mathbin{[]} \mathsf{c} \rightarrow \mathsf{Q}'.$

There isn't any "synchronization" between the now aliased a and b events. If a was a procedure call, and b guarded a procedure definition, the alias would not result in P calling itself, as we would want.If there are two processes in parallel, one that initiates a procedure call and one that executes the procedure body, we expect that once the procedure call is initiated, the process initiating the procedure call will wait while the procedure body is executed. Then, once that is complete, the initiating procedure will continue. If, in the process P above, we consider a to be the initiation and b to be the signal to start the procedure body executing, we would expect the event c (now the combined "start-of-procedure" event) to lead to the execution of Q′, the procedure body, followed by R′, the rest of the program. Instead, one of the two will occur (but we don't know which, since the choice is non-deterministic), and the other will not. Rather than the alternative between R′ and Q′ above, we would rather have interpreted the aliasing of a and b in P with a construction such as

$\mathsf{P}' = \mathsf{c} \rightarrow (\mathsf{Q}';\mathsf{P}')$
or
$\mathsf{P}' = \mathsf{c} \rightarrow (\mathsf{R}' \| \mathsf{Q}')$

However, it is not clear how to create a general form for synchronizing events within a sequential process, which is why the construct has been omitted from the general language definition. The problem of how to describe events separately in a sequential model and then to combine them is similar to problems associated with describing different views of the same event. The difficulties of structuring views and actions in a sequential model are discussed in, for example [Jac95a].

While explicitly constructed self-loops (such as a recursive procedure call) may be avoided in many styles (it is simple to define a constraint that prevents them), they factor significantly in the intermediate forms of structural induction proofs such as those demonstrated in chapter 5. Thus, we felt it was better to omit direct component-component synchronization altogether, rather than to complicate reasoning in WRIGHT through the frequent occurrence of special cases. An alternative might be to provide a completely different, explicitly synchronous semantics to connection. We discuss the issue of alternative semantic models for architectural description in future work (chapter 10).

9.1.2 Why not CSP Directly?

As we saw in chapter 3 and in the case studies, WRIGHT makes heavy use of the semantics of CSP. The question therefore arises, why not simply use CSP directly to describe architectures? Why do we need an explicitly architectural notion at all? We believe that there are important methodological reasons why it makes sense to provide a specialized notation for architectural specification.

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

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

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

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

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

9.1.3 Dynamism

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

We believe that the choice to limit WRIGHT to static architectures is the right one. First, the class of non-dynamic systems is both significant and complex. For these systems, including dynamic features complicates analysis, because the structure of the architecture cannot then be used to guide reasoning about the behavior of the system. If we cannot assume that the structure is fixed, then we must consider all possible structural variations as well as all possible behavior variations. We believe that separating these concerns and focussing WRIGHT on abstract behavior in a fixed architectural configuration provides significant leverage by reducing the complexity of analysis.

Second, many systems that are commonly thought of as a single, dynamic architectural configuration can be treated as a specialized architectural style. The style would have tight topological constraints and use parameterized components and connectors. For example, a client-server system with a varying number of clients on different executions could be considered "dynamic," with different clients arriving and departing unpredictably. However, on any given execution of the architecture there will be a fixed number of clients. Thus, we can define a server type that is parameterized by the number of client connections and define a style that covers each of the topologies of interest. Then, any execution will have a static configuration with a fixed number of clients. All of the mechanisms we have described above for WRIGHT can be applied directly within this framework: Any given execution can be analyzed as a configuration with a given behavior; if we wish to establish a property for all possible configurations, we can reason about the style as a family of systems.

Third, a common case of dynamism is in systems with setup and take-down phases. That is, many systems have three distinct execution phases. First, components are launched and communications paths are initialized. Second, there is a "normal" execution phase, during which the configuration of the system is fixed. Third, there is a final phase in which communication is halted and components terminate.

These systems can be handled in WRIGHT. One way to handle these systems is to ignore setup and take-down. Using this technique, only the normal execution phase is considered. The normal execution phase is the phase during which consistency is most important. The other phases are responsible for getting into and out of that consistent state. The bulk of any complex computation will usually fall into the normal phase.

Another way of modelling setup and take-down in WRIGHT is to treat the architecture as existing at all times, and to view setup and take-down activities as just more events on the existing communication paths. This is how setup and take-down is handled in both the AEGIS (chapter 4) and HLA case studies (chapter 6). In HLA, federates that join or resign from a federation execution are treated as existing at all times, but use joinFedExecution and resignFedExecution events to signal their start and finish. The protocol is defined so that no normal activity is permitted when the component logically should not be executing. Using this kind of model, we are able to analyze not just the activity of a configuration while it is in a fixed state, but also during setup and take-down. In the HLA example, we were able to show a race condition between creating the RTI connector and having a component attempt to locate it.

Finally, we claim that WRIGHT can contribute to the analysis even of systems where the dynamic character of the architecture is inseparable from its abstract behavior — when the architecture must change during normal execution in order to achieve its goals, or where there is no way of bounding the size of the architecture even on a single execution. Even for these systems it is important to describe and analyze the behavior of components and their interactions during those times when re-configuration is not taking place.

Finding an appropriate formal basis for describing the dynamic aspects of architecture, as well as integrating the kinds of behavior analysis that WRIGHT supports with those descriptions, is a significant possible avenue of future research (see chapter 10).

9.2 Description of Styles

9.2.1 Use of Constraints

A major feature of WRIGHT is its definition of architectural styles as predicates that must be satisfied by all instances of that style. This differs from previous treatments of style definition [AAG93, MQ94, GAO94] by focussing on properties that must be satisfied by instances of the style rather than on the definition of unique vocabulary, semantic models, and tools for a style.

To elaborate, the style definition models of Abowd *et al.* [AAG93] and of Moriconi *et al.* [MQ94] are based on the idea that a style defines a collection of syntactic structures, a semantic model for reasoning about systems, and a mapping between the two. While Abowd *et al.* do define a generic *abstract* syntax for architectural configurations, they emphasize that a significant element of a style is its unique concrete syntax. That is, syntax is developed for each style with no necessary relation with the vocabulary of any other style.

Additionally, the semantic model of an architecture (*i.e.*, how meaning is assigned to a component or connector, and how components and connectors are combined into a configuration) must be constructed for each style. While Moriconi *et al.* do permit the definition of axioms (*i.e.*, constraints) that must hold of a configuration in a style, their model is explicitly separated from any tie to a particular description language or semantic model. Instead their model requires that a style definition include both a syntax and a semantic theory in which reasoning about systems will occur. That is, in this model axioms can restrict a style beyond the full scope of the semantic theory but do not form a sufficient definition of the style.

An Aesop architecture environment [GAO94] takes a somewhat different approach. Rather than supply a semantic theory in which to reason about architectures in a given style, a style definition consists of a collection of type definitions, which are subtypes of the basic types of component, connector, configuration, *etc*. The semantics of the style are embedded in the methods of the types. These methods are used to manipulate designs in the process of being created, that is, for evolving an architecture description. Thus, the scope of a style is defined by the closure of the operations that can be applied in sequence, beginning with an empty (or other initial supplied) architecture description. The semantics of these systems are not explicitly defined, but rather are implicit in special methods that take advantage of the described systems properties. For example, a "build-me" method might be defined for a configuration. This method would produce an executable implementation of the architecture. Thus, the "build-me" method defines the behavior of a given architecture description by converting the description into another artifact that has a defined behavior. The approach of defining semantics through methods on types provides no explicit means of determining whether a given configuration is a member of a style, making that information implicit in the vocabulary used, and leaves the interpretation of that vocabulary up to developers who build tools to take advantage of the style.

While a WRIGHT style can introduce vocabulary through the definition of types, it is not required to do so. Further, the semantics of configurations are consistent across all descriptions, regardless of style, and a configuration description may introduce and use its own types beyond those provided by the style definition.

As we argue below, the advantage of our approach is two-fold: First, it means that a style is characterized directly, rather than implicitly by the range of systems that are describable using the mechanisms provided. Second, the use of a common language for the description of configurations means that it is easier to define and use a new style.

9.2.1.1 Direct characterization of style

We have argued that an important benefit of formalizing architectural style is the ability to reason about its members as a family of systems. In chapter 5, we showed how reasoning about a style could provide important guarantees to users of that style. It was because of a direct characterization of that style that we were able to do so; the constraint predicates of the style provide a set of axioms for reasoning about the systems that are in the style. If we are concerned about a given property, such as global-deadlock or data-consistency, we can we can reason directly from the style definition to prove formally whether the property must hold for all members of the style, does not hold for any member, or holds for some and not others. Proving the property for a given system is then a simple matter of evaluating the style's defining constraints on the configuration: if the predicates are true, the system is in the style and the general proof applies directly.

If, on the other hand, as is done in other approaches, a style is defined solely in terms of a style-specific vocabulary and tools (either development tools or semantic models) that are applied to systems in that style, it is much less straightforward to determine the common properties of the style. It is generally not possible to directly characterize a system that may or may not be a member of the style; rather, only those that are members of the style can be characterized, and the inability to state something in the given vocabulary represents "non-membership." Proofs regarding the ability of a language to express something are difficult, requiring the introduction of meta-languages and characterization of the power of the reasoning system introduced by a given semantic theory.

The inability to distinguish the semantic theory of a vocabulary from a family of systems leads to complications in reasoning models about systems and styles. For example, Moriconi *et al.*'s models of architectural refinement require a completeness assumption about any specification. That is, it must be assumed that not only are all provable properties true, but all unprovable properties are false. The completeness assumption means that all proofs of refinement or style membership are required to make an argument about the weakness of their specification as well as the much simpler demonstration of property satisfaction. This further means that if an architect wishes to permit a property to be un-specified in a given system description, he or she must select a semantic theory in which it is *impossible* to state that property. Otherwise, failure to specify a property supplies a default answer.

In the Aesop system, the use of vocabulary to delimit the coverage of a style means that Aesop is forced to define a non-standard interpretation of subtyping. Typically, a subtype of another type is required to preserve the semantics of any methods. That is, if two types s and t share a method m, and s is a subtype of t, then the effect of applying the method m to an object of type s is required to be the same as the effect of applying that method to an equivalent object of type t. In Aesop, this requirement is weakened to include the possibility that the method may *do nothing* even when there was an effect on the object of type t. Thus, whenever a tool applies a method to any object, it must then check whether an effect actually occured, in case a subtype was involved.

In each of these cases, reasoning about a system is significantly more complicated than necessary. In the model of [MQ94], the architect is constantly forced to reason about all of the things that *may not* be specified, as well as those that must be specified. In Aesop, establishing that a system is in a given style does not provide sufficient information about what can be done with it, because a sub-style might reduce the number of legal manipulations. Further, establishing that a configuration is in a style consists of finding a sequence of methods (of unknown effect) that will result in the construction of that configuration.

9.2.1.2 Use of a common language and semantics for configurations

In this thesis, we have argued that an important goal is to provide a practical basis for the definition and use of architectural styles. One way that we have achieved this goal is by supplying an architecturally structured semantic model for configurations. That is, by describing an architecture in WRIGHT, it becomes possible to reason about the properties of that system, and to use the architectural structure of the description to structure analysis. The achieve this, the semantics of WRIGHT defines mappings from local to global names (of events), instantiation of multiple instances of a single type, and the compositional behavior of interacting components.

A style definition in WRIGHT can take advantage of that architectural structure directly, because all descriptions of architectures use that structure and the underlying semantics, and the constraint predicates can refer to the structure directly.

If a style description provides its own semantic model, as is done in [AAG93] and [MQ94], then the style developer is forced to redo all of the architectural structuring of the semantics provided by WRIGHT. This is a non-trivial undertaking and significantly increases the cost of creating a style. Thus, a style would only be developed if there is a relatively high potential payoff, and then none of the effort of developing one semantic model can be reused in another style.

In WRIGHT on the other hand, the cost of specifying even a very specialized style is relatively small. It is convenient to specify a style that is just constrained enough to enable an analysis or to guarantee a property of interest. Further, simply by including the vocabulary and defining predicates in another style, all of the analytic results of one style can be transferred to another.[2] Thus, architects can better take advantage of formal definition and analysis of architectural styles.

9.2.2 Are Constraints at the Right Level of Power?

One property that the definition of styles through special-purpose vocabulary or types has is that it is *constructive*. That is, the definition of the style includes directly the means of building a configuration that will be guaranteed to be in the style. WRIGHT's use of constraint predicates, on the other hand, makes membership of a configuration in a style a *proof obligation*. That is, one must first construct the configuration and then demonstrate that it is a member of the style.

This is a potential problem because one can write constraints that are arbitrarily difficult to satisfy. Consider, for example, the JC style described in chapter 5. In that chapter we defined a predicate $Deadlock$, as follows:

$$\begin{aligned} &Deadlock \mathrel{\widehat{=}} \exists\, t : Traces(System) \\ &\quad \mid (t, \alpha System) \in Failures(System) \wedge last(t) \neq \surd \end{aligned}$$

We can legally define a style in which the sole constraint is $\neg\ Deadlock$, that is, that the system must be globally deadlock free. But this provides *no help whatsoever* to developers of systems. It is obvious on the face of it that we want to build a system that does not contain global deadlocks — the hard question is how to accomplish that.

The question therefore arises: Is there a way in which we could simplify or restrict the constraint language to guarantee that styles will be constructive, *i.e.*, that the definition of a style makes it clear how to construct a system that is a member of the style?

Unfortunately, it is not reasonable to expect that we can create a language of predicates such that it covers only systems that are easy to construct without unduly limiting our ability to define styles of interest in real-world situations. Indeed, balancing the architect's desire for construction guidance with the need for analytically powerful styles is the essence of the style design problem. To discover or design a class of systems that share a given property and that are easily constructed is fundamentally a creative task, and one that we do not aspire to solve through formalism. Rather, we hope to provide the style designer and the system architect with the means to express their solutions in such a way that they can be checked for consistency and completeness and that formal tools can be applied to them.

[2] Of course, the two styles' predicates must not contradict each other. In that case, the resulting style contains no instances.

With the goal in mind of making available analytic and expressive tools, we have chosen not to arbitrarily restrict the constraints that may be placed on a style. Instead, we have included enough expressive power so that style designers can find their own solutions.

9.3 Analysis of Configurations and Styles

9.3.1 Why CSP?

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

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

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

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

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

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

Possible alternative formal bases?

This is not to argue that CSP is the *only* notation that could have been used for WRIGHT. Indeed, we recognize that the choice of CSP allows us to capture only a certain class of properties. For example, WRIGHT does not handle properties such as timing behavior of interactions or fairness because CSP's semantic model is not rich enough. As we saw in chapter 8, it is possible to retain the general descriptive framework of WRIGHT, but replace CSP with an alternative formalism. This argues that the notion of an ADL tied to a particular formalism can be generalized beyond the specific example of WRIGHT, and used to support other kinds of analysis of complex software systems. Further, by retaining a common structural framework, we can compare and combine results from different formalisms.

9.3.2 Initiated v. Observed Events

9.3.2.1 Why make the distinction?

WRIGHT departs from the basic CSP notation through the addition of a distinction between initiating and observing an event, which is indicated by adding an overbar ($\overline{e}$) to initiated events. Recall that, as we described in chapter 3, every event in WRIGHT has one party (a component or a connector) that initiates the event, causing it to occur, while all others observe the event. One might well ask whether this distinction is necessary.

As we argued previously (in section 9.1.1.1) regarding the need to explicitly and locally describe connectors in WRIGHT, it is important to recognize that there are issues and constructs, that, while general methods may omit them to achieve formal simplicity, are of vital importance to the task of designing and analyzing a complex software system. One of these is, we believe, the concept of *locus of control*. While CSP's model is excellent at describing the collection of possible sequences of events, and for locating conflicts regarding the ordering of events, its model of compositionality deliberately hides the substructure of a process. That is, once two processes have been composed via $\|$, there is no formal distinction between the composed process and one that has the same behavior but was constructed solely using sequential operators ($\rightarrow$, $\Box$, *etc*). In fact, this equivalence is the essence of a process-algebraic model of composition. There are rules for describing the meaning of parallel composition in terms of the sequential operators. Thus, in CSP it is not possible to distinguish what subprocess "caused" the event. In a composed process, the event occurs globally, and represents an external observation of the interaction, rather than an action by some party that is now internal to the process.

On the other hand, software architectural analysis is specifically focussed on how different, identifiable parts behave when composed, and on the construction of components that will *cause* a certain effect to occur. That is, when considering the behavior of an architecture, we do not only care what happens. Rather, we also care what component is responsible for any given event. We must distinguish between a component that ensures that an event will occur and one that simply admits that an event might possibly occur. Thus, we have chosen to add the initiated/observed distinction to the basic semantics of WRIGHT. This has led to new opportunities for analysis and checking, as we discussed in previous chapters.

9.3.2.2 Initiator commits too strong

Recall that one of the basic consistency checks in WRIGHT is the rule of initiator commits. This check ensures that a process that claims to initiate an event does indeed treat the event appropriately, not permitting outside control of the event at the moment of decision.

As we discussed in chapter 3, there are situations where a specification can be considered consistent even if it does not obey the strict definition of the "initiator commits" rule. One such situation is when a component contains internal concurrency. Another situation is when the continuation of a protocol depends on distinguishing different return values from a procedure call.

The first situation, internal component concurrency, can occur in two ways. First, a component may be implemented directly using a threading mechanism, as we saw in the AEGIS example of chapter 4. Second, whenever a WRIGHT component is decomposed hierarchically, the process defining the composed component contains internal concurrency. That is, each of the components contained within the larger component are represented as distinct threads within the larger component. In each of these cases, race conditions between different threads that initiate events will appear as violations of the basic initiator commits rule. The rule can still be effectively applied, but it must be applied to each thread within the component rather than to the complete computation. In the case of a hierarchical decomposition, the rule applies only to the components at the lowest level of the hierarchy. In the case of a non-hierarchical WRIGHT computation that contains concurrency directly, the analysis must be tailored to the particular form that the concurrency takes.

The second situation is when a single logical interaction event is split into two different events. For example, recall the Pipe connector that we used as an example in chapter 3. The Sink role, when it chooses to receive data, must be aware of the possibility that it may be notified of end-of-data instead. We are forced, under the initiator commits rule, to divide the call and result of a pipe into two parts:

$$\textbf{Role}\ \mathsf{Sink} = (\overline{\mathsf{read}} \rightarrow (\mathsf{data?x} \rightarrow \mathsf{Sink}\ \square\ \mathsf{end\text{-}of\text{-}data} \rightarrow \overline{\mathsf{close}} \rightarrow \S)) \sqcap (\overline{\mathsf{close}} \rightarrow \S)$$

The first part of the protocol is a notification to the **Glue** that a value is desired (via event $\overline{\mathsf{read}}$), and the second is either a data value (data?x) or an end of file (end-of-data). The problem with this specification is that we have, in effect, divided what should be a single event, a procedure call, into multiple events. While this may be a perfectly accurate characterization of the internal behavior of the system, it has the effect of *lowering* the level of abstraction in this connector. The description now shows a single concept, the read procedure, as two concepts, request and result.

What we would have preferred to see is the following:

$$\textbf{Role}\ \mathsf{Sink} = (\overline{\mathsf{read?x}} \rightarrow \mathsf{Sink}\ \square\ \overline{\mathsf{read\text{-}end\text{-}of\text{-}data}} \rightarrow \overline{\mathsf{close}} \rightarrow \S)) \sqcap (\overline{\mathsf{close}} \rightarrow \S)$$

In this formulation, the return value on the data is used to distinguish control paths, by distinguishing a normal value (?x) from a special result (-end-of-data, which is now effectively a value rather than a separate event). In this case, read is really thought of as one event, which is appropriately initiated (and committed) by the role, but for which the **Glue** will decide what data value it will take. If we permitted the use of embedded conditionals, we could get a similar effect by treating eof as a special data value and making the behavior conditional on x:

$$\textbf{Role}\ \mathsf{Sink} = (\overline{\mathsf{close}} \rightarrow \S) \sqcap \overline{\mathsf{read?x}} \rightarrow \begin{cases} \overline{\mathsf{close}} \rightarrow \S, & \textbf{when}\ x = eof \\ \mathsf{Sink}, & \textbf{otherwise} \end{cases}$$

Unfortunately, this solution is also not permitted by the current form of our initiator commits rule. The problem here is that CSP treats all events as atomic entities, with no data component. That is, read?x is really a family of events, read.1, read.2, *etc.*, and read-end-of-data is just another event. read-end-of-data has no formal tie to the other read events, and in fact, there is no formal way of tying any of the events read.1 *etc.* to each other. Thus, in our earlier definition of "initiator commits" there is no means of indicating that the read events and read-end-of-data are really "one" event and that they should be grouped together for the purposes of the test.

This is a real weakness of the initiator commits rule in WRIGHT. It means that a failure of an architectural element to pass the initiator-commits test must be considered by the architect on a case-by-case basis, taking into account the possibility of concurrency and the need to balance level-of-abstraction with accuracy. However, we believe that there is significant value in identifying situations where there might be a problem with an element failing to treat an initiated event properly, and that this justifies including the test in the standard suite of checks.

9.4 Relevance to Practice

Throughout this dissertation we have argued that WRIGHT represents a practical approach to the description and analysis of software architecture. By providing abstractions that match those used by architects, by defining simple, automatable checks that can locate common and significant errors in architectures, by supporting the analysis of families of systems through style, and by providing a precise, analyzable basis for characterizing the abstract behavior of architectures, WRIGHT represents an important tool for the software architect. We have also shown, in particular in chapter 6, how WRIGHT's descriptive mechanisms can be combined with specific abstraction and localization techniques to support the incremental analysis of complex, real-world systems. Thus, we have demonstrated that WRIGHT scales to support precise, formal, description and analysis of interaction patterns in large-scale systems.

But the ability to scale is not sufficient to guarantee that WRIGHT can be applied in practice. In addition, the use of WRIGHT must fit into the general context of software engineering practice. It must be cost effective to apply WRIGHT in a resource-constrained environment, the architectural effort must provide benefits at other stages of the software lifecycle, and it must be possible to integrate the formal, architectural effort with existing software design practices.

In this section we address three specific issues regarding WRIGHT's relevance to software engineering practice. These issues are the cost of using formalism in engineering, WRIGHT's relation to a system's implementation, and the application of WRIGHT's techniques to non-architectural models of design.

9.4.1 Can Formalism be Effective in Software Engineering?

In the introduction to this dissertation and in section 9.1.2 we argued that basic, general-purpose formal notations such as CSP and Z, in their raw form, are not well suited to the problem of software architecture and analysis. It is too cumbersome to describe a complex system in these general formalisms, requiring the reinvention of mechanisms for localizing components and connectors and for their composition. Further (as we argued in section 9.2.1.2), using a general formalism to describe styles does not avoid the problem of re-inventing structures for configurations, because it is still necessary to invent a special vocabulary and semantic model before any results can be applied to instances of the style.

We argue that WRIGHT overcomes these problems by structuring the formalization task to be directly applicable to architectures, by increasing the applicability of analysis through architectural style, and by supporting the incremental development and analysis of architectural descriptions.

First, WRIGHT provides a notation in which the structure of the formal models of behavior match the structure of large systems. Components and connectors are distinguished, and they may be described by reusable, independent types that can be flexibly combined in various configurations. Rather than requiring the analyst of a particular system to develop special mechanisms for relating the local descriptions of individual computations and interactions to the global behavior of a composed configuration, WRIGHT provides the appropriate models directly. Thus, because WRIGHT is tailored to the task, it is easier to describe and analyze an architecture in WRIGHT than in general purpose formalisms.

Second, by supporting the description of families of systems as styles, and by supporting the application of style-based analyses to actual configurations, WRIGHT permits the localization of formalization effort in a development enterprise. As we saw in the JC and HLA case studies (chapters 5 and 6), it is possible to analyze entire classes of systems that share a common vocabulary and obey simple topological rules. Once a style has been formalized, the results from that style's formalization can be applied directly to configurations that are derived from the style. If the appropriate vocabulary is used, no further formal effort is necessary beyond evaluating the topological predicates.

For example, any analysis performed on the HLA style of chapter 6 is directly applicable to *every* federation developed. There is no need for any developer of a federation to perform formal analysis, or even to understand the already-completed formalization. The correspondence between the English language standard and the formal description has been spelled out by the style analyst, and the federation developer can therefore refer solely to the standard, while at the same time taking advantage of the results of formalization in WRIGHT.

Third, WRIGHT's support for incremental development of descriptions, partial descriptions, and selection of the level of detail mean that some benefits of formalization can be achieved at a low initial cost. If an architect does not want to take the time to fully define the protocol of a particular interaction, the DefUse connector described in section 9.1.1.2 can be used immediately.[3] Even with this simple connector, the architect can determine whether the routines called by one component match up with those defined by another, without needing to construct the entire system and to attempt to compile it.

Then, if the architect has more information about a particular interaction, or wishes to explore a particular question regarding the interaction of different call orderings, he or she can substitute another, more detailed connector for DefUse, and thereby deepen the analysis.

[3]But see that section for a discussion of its limitations.

For example, if some of the routines in an interaction are specifically related to initialization, the roles can be modified to require that those routines are used before the others, and, once initialization is complete, the rest of the interaction can be left unconstrained. WRIGHT's analytic capabilities can be applied to this slightly more detailed protocol without requiring a full elaboration of the protocols following initialization.

We have shown through the HLA case study how this incremental approach can be pursued further, successively deepening the analysis through the elaboration of a protocol, each elaboration requiring only a level of effort that is commensurate with the benefits to be gained from the newly enabled analyses.

9.4.2 Relation to Implementation

One thing that distinguishes WRIGHT from many other architectural notations is that it does not explicitly state how the architectural description is related to a system's implementation. Instead, WRIGHT provides an abstract description of system behavior without reference to particular implementation constructs. An important question, therefore, is how an architecture description in WRIGHT can be related to an implementation.

There are several techniques that may be used to relate a WRIGHT description to a more concrete level of detail. These include automatic association of implementation constructs with abstract descriptions and maintenance of a correspondence through style-specific libraries and compilers.

9.4.2.1 Automating association to implementation

An existing body of work addresses the question of how, using special annotations, one can extract a finite-state-automaton specification from a program written in Ada and perform analysis on it [CA95]. There is currently work underway on automatically transforming WRIGHT specifications into these FSA-machines for the purpose of verifying the correspondence between a WRIGHT specification and an Ada program. This technique has successfully been applied to specifications of some "model" problems, such as the gas station pump [NACO97]. This work is closely related to the general question of refinement in architecture, which is discussed in future work.

9.4.2.2 Maintenance of association through style

Another approach to associating implementations with WRIGHT specification is through the recognition that many styles have specific implementation techniques associated with them and, in particular, are often constructed around infrastructure to support particular connectors. This kind of correspondence between abstract connectors and style-specific infrastructure, that supports a one-to-one correspondence between abstract components and implementation modules, is common. It is often, as we discussed in the introduction, a motivating factor behind the description of an architectural style.

For example, the HLA case study discussed in chapter 6 is an analysis of a standard that is specifically designed to support infrastructure for communication between elements of a simulation. In this case, each event in the style's single connector type, the RTI, corresponds to a specific message that can be sent or received by the run-time infrastructure of systems constructed to meet the standard. Thus, as we discussed in that chapter, there is a clear correspondence between the structure of the WRIGHT architecture description and the structure of any implementation using the High-Level Architecture, and a similarly clear correspondence between abstract behavior described by the WRIGHT specification and the actual behavior of the implementation.

While there might not be a formal definition of the relationship between a WRIGHT specification and an implementation, for the simple reason that there is often no formal definition of the implementation, we would argue that in most cases, especially when a style is used, the results of a WRIGHT analysis are directly applicable to the implementation of the described system.

9.4.3 Integration with Other Design Techniques

A significant issue for the effective use of WRIGHT in practice is its ability to integrate with design models that are not explicitly architectural. There are many design practices, such as object-oriented design, in common use, and if using WRIGHT involves discarding these techniques and losing their benefits, there is little hope for the adoption of WRIGHT in practice.

While we cannot, obviously, cover all existing models of design, in this section we show how WRIGHT can be integrated with a particular, common method, object-oriented design. We further show how use of WRIGHT's descriptive mechanisms can enhance both current models of object-oriented typing and emerging techniques such as object-oriented design patterns. We show WRIGHT's compatibility by describing how a particular kind of architectural description can model object-oriented types, and thus an object-oriented design.[4]

In our discussion of syntactic shortcuts in WRIGHT, we described a simple DefUse connector, which defined a bundle of routines that would be provided by one party and called by another (section 9.1.1.2). This is exactly the abstraction presented by an object-oriented type. A slightly more complex version of DefUse can be interpreted to represent an object-oriented type. This version would include parameter and return value types on the request and reply events respectively.

We model data parameters in WRIGHT by extending DefUse to define sets of values that are associated with parameter and return values. Data values are accepted by the Definer through a deterministic choice over versions of request ($\Box\ v : T \bullet \mathsf{request?}v \rightarrow \ldots$). Return values are chosen by a non-deterministic choice ($\sqcap\ v : T \bullet \overline{\mathsf{result!}v} \rightarrow ..$). For the User, parameter choice is modelled by non-deterministic choice and return value is modelled by deterministic choice.

We can view the component playing the Definer role as the implementation of the type, while the component playing the User role is a client of the type. According to WRIGHT's semantics, the behavior of the configuration is that each component that plays a User role invokes the methods of the Definer based on the internals of its computation and on methods

[4]Of course, as we discussed earlier, WRIGHT is limited to systems that do not depend on arbitrary reconfigurations. Many object-oriented designs are relatively static, however, and the techniques here apply in those cases.

that are invoked on it (through the component's other ports, which play Definer roles in other interactions). The component playing the Definer role responds to the requests, carrying out a computation that changes the component's internal state and calculates return values, which are then sent back to the User via the result event. Thus, the behavior ascribed to the WRIGHT configuration matches that of an object-oriented description of the system when each component corresponds to an object instance and each DefUse connector corresponds to a reference from one object to another.

Further, under this interpretation WRIGHT's port-role compatibility checks match the standard object-oriented type "compatibility" checks. Any component attached to the User role has the opportunity to call any of the methods defined, and if it does so, it must provide parameters (to the request event) that do not fall outside the range of permitted values. Also, it must then respond to the result event, accepting all possible return values. In order to be compatible with the User role, the component is not *required* to call any of the methods that are offered, but it may not call any that are not offered. Thus, port-role compatibility for the User role matches the type checks on proper use of a declared type.

The criteria for a port's compatibility with the Definer role also corresponds to object-oriented compatibility checks, in this case to subtype conformance. A component playing the Definer must respond to all of the methods (via the request event), and handle all possible parameter values supplied (*i.e.*, all values of the parameter's type). In responding to the method invocation, the component must provide a return value (via the result event) that is within the defined range, but need not cover the entire output range. That is, the type of the result must conform to the declared type, but may actually be a subtype. However, the component filling the Definer role may also offer additional methods, and further, may accept parameter values on the request event that are not specified by the Definer role. The Definer component is, *only in situations where the parameters are outside the range of the role*, permitted to expand the range of possible return values beyond those defined by the connector.

These conditions for when one type (the component's port interface) is compatible with another type (the connector's role interface), are exactly the same as the constraints defined for accepted models of contravariant subtyping in abstract datatypes (as discussed, for example, in [CW85]). Thus, the combination of the DefUse connector and WRIGHT's port-role compatibility check quite naturally model object-oriented types.

Therefore it is possible to construct an architectural description (in WRIGHT) that is directly comparable with an object-oriented design of the same system. Further, once an equivalent architectural description has been established, we can use the incremental development techniques discussed above (section 9.4.1 and chapter 6) to extend the architectural description to take advantage of the full descriptive and analytic power of WRIGHT, while retaining the correspondence with the object-based system design description. That is, the dynamic information provided by WRIGHT's descriptive mechanism can be used to extend the basic static notion of typing.

There is current research in the object-oriented community to do similar kinds of things. For example, Liskov and Wing [LW94] have presented a view of a type as defining a behavior as well as an interface. They use this to extend the idea of subtyping from static interfaces to "history" properties, requiring that a subtype's new methods be defined in such a way that they do not extend the possible time-sequences of states of the object. One way of understanding the abstract state of an object is to restrict the allowable return values of methods. That is, if we know an object's state at a given time, we can predict how it will respond to a given request, *e.g.*, to provide the value of one of its attributes. Thus, if we construct a specialized interface to describe how a given sequence of requests will alter the permitted return values, we have defined how the methods alter the abstract state of the object. Having made this extension to the interface, the port-role compatibility match also supports a history-dependent model of subtyping, because the protocol defines permitted sequences of states, and the protocol must be respected by any compatible port.

While one could use WRIGHT to construct such a default notion of permitted sequences, WRIGHT also permits the architect to explicitly specify the allowable sequences of events. Thus, one can explicitly state the constraints that any user of the type must obey and the assumptions that a user is permitted to make about the object. That is, the use of *explicit* history constraints (rather than the implicit constraint of Liskov and Wing) permits the definer of a type to clarify exactly what constraints there are on the behavior of the object. By modifying the constraints, one can permit more subtypes when there are weaker constraints and fewer subtypes when there are stronger constraints.

The notion of treating object types as protocols has been pursued directly in the object-oriented context, by Nierstrasz [Nie93]. In that work, Nierstrasz develops a model of subtyping based on protocol conformance that is similar to port-role compatibility. (See chapter 2 for more details.)

Beyond simply matching existing work based on a notion of an asymmetrical definition/use interaction, however, WRIGHT connectors also support a more symmetrical model of interaction than that available through current object models. Although we do not directly demonstrate it in this thesis, this kind of symmetry would be useful in the analysis of design patterns, as defined, for example, in [GHJV95]. Patterns are intended to capture design expertise in reusable frameworks by defining a pattern of object collaboration, constraints on context of use, and a characterization of the problem solved by the pattern. In the object patterns, multiple objects are described that each must use the other objects according to the pattern of collaboration. Thus, the objects' relation is symmetrical, because two (or more) objects refer to each other, and calls back and forth must be coordinated to satisfy the computation's needs.

Currently, design patterns are defined informally, using OMT diagrams [RBP+91] of one example of the pattern's use, with additional text to explain which parts of the example are variable and which fixed. It would be a valuable addition to the design pattern model to be able to describe a "pattern of object collaboration" more precisely. Patterns could then be presented as a set of systems, with an explicit, formal, specification of the variability in the set as well as the ability to precisely constrain the permitted behavior of both the fixed elements and variable elements of the pattern.

This is exactly what WRIGHT's style definitions provide for software architecture, and it seems reasonable to expect that WRIGHT's models could be used as a starting point in providing a more rigorous basis for design pattern description. In particular, the models of interaction supported by WRIGHT connectors are both compatible with object-oriented typing (as we have discussed) and provide a more flexible means of describing the constraints on the interaction between design elements.

Chapter 10

Conclusion

10.1 Summary

In this dissertation, we have shown that:

> *An Architecture Description Language based on a formal, abstract model of system behavior can provide a practical means of describing and analyzing patterns of discrete, asynchronous interaction in software architectures and architectural styles.*

This was demonstrated through the example of a particular formal Architecture Description Language, WRIGHT, which models connector types as abstract patterns of interaction as relationships between discrete, asynchronous component actions, and defines architectural styles by predicates constraining configurations. In chapter 3 we introduced WRIGHT, its syntax, semantics, and consistency and completeness checks. This chapter laid the groundwork for our demonstration, showing how a formal notation can be developed that provides a direct model of architectural computations and interactions in a form that can be used both to describe and to analyze configurations and styles.

Then, in chapters 4 through 6, we explored the expressiveness and practicality of WRIGHT through three case studies. The first, AEGIS, demonstrated the utility of precise definition of architectural configurations, and showed how the abstract description of behaviors exposes critical issues and contributes to the effective consideration of architectural design alternatives.

The second case study presented the formalization in WRIGHT of an architectural style, the Justo-Cunha Style, and showed how explicit characterization of an architectural style, supported by WRIGHT's formal basis, structures analysis of that style. Through the JC case study, we showed how use of WRIGHT to characterize a style enables us both to simplify reasoning about members of the style and to provide assurances that all members of the style have specific, critical properties, such as global deadlock-freedom.

The final case study, of the "High-Level Architecture for Simulations" demonstrated that WRIGHT's descriptive and analytic facilities could be applied to complex, large-scale, real-world systems. Through our formalization of the HLA style, which was part of the effort to develop a draft standards document, we were able to find critical ambiguities and inconsistencies in the draft specification. In this case study we also introduced and demonstrated specific techniques for the incremental description and analysis of large systems, allowing us to balance the effort of formalization against the benefit to be gained. Thus, we showed how WRIGHT provides a practical basis for the formalization of real systems.

In chapter 7 we discussed the automation of analysis in WRIGHT. In chapter 8 we provided an example of how properties that are not easily characterized in event oriented notations such as CSP might be expressed within a WRIGHT-based framework, by creating abstract specifications of those properties (in Z, in our example) and defining correspondences between the alternate formalization and a WRIGHT specification. We thus showed how the architectural structure of a formal ADL might help support formalization of properties even when those properties are not directly expressed in the underlying semantics of that ADL.

Finally, in chapter 9, we considered WRIGHT in light of the question of whether it satisfies the requirements for a practical basis for analysis of architectural configurations and styles. We thus concluded that, while there are alternatives to the design decisions made in this dissertation and ways in which it could profitably be extended, WRIGHT does indeed provide a practical basis for description and analysis of interactions and styles in software architecture.

10.2 Contributions

The primary contribution of this dissertation is its role as an existence proof for the practical formalization of architecture. By providing an example of how a particular aspect of software architecture, patterns of interaction, can be formalized and exploited through an architecture description language, we have shown that a formal approach to software architecture can be both valuable and practical.

Additionally, this thesis makes contributions in four main categories: the general foundations of software architecture, description and analysis of software architectures, the value of WRIGHT specifically, and formal methods in general.

10.2.1 Contributions to Architectural Foundations

There is a growing body of work on defining the foundational concepts of software architecture, including the abstractions of components, connectors, and style, and on the relation between these elements and the properties of systems. Through our treatment of these abstractions in a formal, coherent notation, we have contributed to the understanding and precise characterization of foundations of software architecture.

- **Explicit Connectors:** Through our exploration of explicitly described connectors in WRIGHT, we have contributed to an improved understanding of a connector as a pattern of interaction. Connectors have been a key, but informal, element of the concept of software architecture almost from the moment "software architecture" was introduced as a term (see, for example [Sha88]). WRIGHT is the first notation to provide mechanisms for the explicit characterization of the semantics of a connector and to define the meaning of that characterization in system composition.

- **Style as Predicate Over Sets of Configurations:** We have provided an analytical model of architectural style as a set of configurations. Previous work on style has either been informal, treating it in terms of shared idioms, or has taken either an implicit or an ad hoc view of style. By characterizing an architectural style directly as a set of configurations within a common semantic framework, this dissertation provides a better basis for direct analysis of styles and their member configurations.

- **Framework for Architectural Semantics Based on Structure:** We have shown how a semantic framework that is explicitly architectural can provide a basis for structuring a model of a system. Recent workshops (*e.g.*, [Gar95, KW96]) have addressed this issue, debating the centrality of a structural view to architecture, and what architecture could contribute beyond a syntactic notion of structure. By providing a structural framework to which semantic models can be tied, we have shown how software architecture forms the basis for modeling the meaning of a system description as the composition of the meaning of its parts. Thus, we have shown how architecture provides a basis for controlling the complexity of a system model by supporting the localization of reasoning when possible, while contributing to an overall understanding of a system's properties.

10.2.2 Contributions to Architectural Analysis and Description

- **Architectural properties beyond structure:** We have presented a model of architectural description in which properties beyond simple structural descriptions can be specified. While previous ADLs have allowed architects to indicate what components and connectors are present in a system, WRIGHT provides mechanisms for describing semantic properties of those structural elements and for analyzing the resulting properties of the entire system. We have thus shown how architectural structure can provide a basis for extending the high-level description of a system to other properties while retaining architectural integrity.

- **Encapsulation:** We have shown how, by structuring the semantics of configurations in terms of the architectural abstractions, we can use the structure of software architecture as a basis for inductive reasoning about entire families of systems. By establishing the property of *encapsulation*, that any configuration in a given style can be treated as an equivalent component of the style, the analytic power of that architectural style is greatly increased.

- **Coverage of instances and styles in same formalism:** We have shown how a single notation can support the description of both architectural configurations and architectural styles. By providing a uniform semantic model and defining styles in terms of sets of configurations, we have shown how analysis can be performed over entire families of systems and yet still be applied to actual systems that are to be constructed.

10.2.3 Specific Benefits of WRIGHT

- **Useful ADL:** Beyond its role as a vehicle for exploring the nature and structure of architectural formalization, WRIGHT is itself a useful tool for architects wishing to describe and analyze their systems. It provides a practical means of precisely characterizing the abstract behavior of an architecture and of analyzing both individual systems and families of systems.

  We also presented techniques, such as localization of internal state, mini-protocols, and selective omission, to support abstraction, traceability, and incrementality in the analysis of large-scale systems. These techniques are generally applicable to WRIGHT specifications and increase our ability both to effectively manage the cost of formalization and to integrate a formalization with the overall development effort.

- **Standard Automated Checks:** In addition to WRIGHT's general analytic capability we have defined a specific collection of checks. These checks can automatically be applied to any WRIGHT description to detect important classes of architectural errors.

- **HLA:** The results from the HLA case study in chapter 6 are directly useful both to guide future revisions of the HLA standard and to help potential developers of systems in that style. Our characterization of the behavior of the RTI provides an important source of documentation both to developers of RTI infrastructure and to developers who want to develop federates and need to understand what behavior they can expect from the RTI. Our analysis also points to issues that need to be resolved either by particular implementations or in future drafts of the standard.

10.2.4 Contributions to Formal Methods in General

While not a primary goal of this thesis, our development of WRIGHT does provide contributions to formal methods in general:

- **Example of Domain Specialization:** WRIGHT is an example of how a general formal method can be used as the basis for a more specialized, domain-specific notation. By tailoring the notation to our domain, software architecture, we have defined analyses that are appropriate to find common problems in the domain. We have further provided a notation that matches the structure of abstractions familiar to domain experts, thus allowing them to express a formal model of their systems in a way that is natural for the domain. At the same time we have taken advantage of the power, expressiveness, and sound basis of a general formal notation by defining our specialized notation in terms of the more general one.

- **Consideration of Families of Systems:** We have shown how, by explicitly considering families of systems, the benefits of a formal analysis can be increased. Once an analysis has been performed, it applies to an entire class of systems, rather than just to a single system. This supports reuse of a formalization, improving the relation of cost to benefit in two ways: First, a given result for a given cost benefits a wider range of systems. Second, expertise in formalization can be directed toward style development, and then the benefits can be enjoyed by system developers who do not need to expend the same effort becoming experts in formal methods.

10.3 Future Work

While WRIGHT represents a significant benefit for software architects, it also suggests a number of areas for possible future research.

10.3.1 Short Term

10.3.1.1 Experience with WRIGHT

Probably the most pressing short term need for research on WRIGHT is to gain experience with its use. In this dissertation we have sampled a variety of different architectures that can be characterized using WRIGHT. With these, we have explored the applicability of WRIGHT and shown how WRIGHT's facilities can be of benefit across a range of systems. However, the case studies described here do not, obviously, exhaust the space of architectures of interest, and they have been carried out by the author in an academic setting. While there has been some use of WRIGHT by people other than the author (*e.g.*, [RD95]), it remains largely untested in actual practice.

The most significant effort currently underway in that regard is an extension of the HLA case study of chapter 6. DMSO is now sponsoring the application of WRIGHT to the current version of the standard along with the extension of the model to cover more details of HLA. Without a doubt, this kind of use will shed further light on how the language can best be evolved to suit the needs of practitioners.

10.3.1.2 Additional resources for WRIGHT users

In addition, developing a collection of syntactic shortcuts for WRIGHT and a library of common architectural styles will greatly increase WRIGHT's usefulness. As we discussed in chapter 9, there are a number of common idioms for connectors (such as name matching on events) for which it would be useful to supply syntactic shortcuts. These can be expressed as process definitions and connector and component types, and it will be important for the effective use of WRIGHT to elaborate these and make them generally available.

Another important resource is to define a library of common and useful styles that can be used by architects developing configurations in those styles. This will greatly increase the value of WRIGHT by making it easier to formalize those configurations.

More experience with WRIGHT is needed to develop these resources. Once these resources are developed, in addition to being useful themselves, they can provide a basis, as we discuss below, for the exploration of important issues such as understanding how to select an architectural style for a given problem.

10.3.1.3 More Automated Tools

As we discussed in chapter 7, there are a few of the WRIGHT consistency checks for which there is currently no automated support. These are "initiator-commits," style constraints, and style consistency. We plan in the near future to develop tools that can provide automated checking of these properties for at least a subset of the WRIGHT language.

10.3.1.4 Coordination with other architectural tools

Another short term need is to coordinate WRIGHT with other current efforts in software architecture. For example, the Aesop [GAO94], UniCon [SDK$^+$95], and Darwin [MDEK95] systems are intended to aid in the construction of systems based on an architectural description. By developing correspondences between Aesop styles and WRIGHT styles, and between WRIGHT connectors and Aesop, UniCon, or Darwin connectors, there would be immediate benefits. WRIGHT would provide analysis tools for systems constructed in these other systems, while they would provide assistance in constructing systems that correspond to a WRIGHT description. Work on this has already been begun, using ACME [GMW97] as a means of merging the different forms of architectural description [WG97].

10.3.2 Longer Term

Beyond using and improving WRIGHT there are several important avenues of long term research suggested by the results of this dissertation. They include the application of alternative semantic models to architecture, the role of architecture in a broader context, the application of tailored formal notations to other problems, and the role of point-of-view in effective large-scale software construction.

10.3.2.1 Alternative semantic models

We have already discussed (in chapter 9) our reasons for selecting CSP as a basis for WRIGHT's semantic model. However, as we also discussed in that chapter, this semantic model limits both the architectures that can be directly expressed (they must be static and asynchronous) and the properties that can be attributed to those architectures (only control and event ordering properties are naturally captured).

We have begun to explore ways in which the descriptive power of WRIGHT can be extended to capture abstractions and issues that are either difficult or impossible to express in WRIGHT. For example, in chapter 8 we showed how Z schemas could be defined to capture a state-oriented property (ownership of data attributes). We further showed how correspondences could be defined between that alternate model and WRIGHT so that the structuring and event-ordering definitions in WRIGHT could inform the analysis of state-properties, while state-properties could inform the control abstractions defined in WRIGHT.

However, that single example goes only so far in answering the general question of how alternative semantic models can be applied to software architecture. For any given class of properties, perhaps for any given class of systems, the notations and models that best suit them will be different. At the same time, it is important not to have to re-develop the common aspects of those models, such as the structuring and compositionality that is provided by the architectural abstractions of components, connectors, configurations, and styles. Issues for alternative semantics include:

- **Semantic Foundations for Architecture:** We chose to use an existing general formal model on which to base WRIGHT not only because it seemed, mostly, to provide the descriptive capabilities that we needed, but also for purely pragmatic reasons. Working out how the architectural abstractions are combined and the kinds of descriptions to provide was difficult enough without the additional task of developing process-algebraic behavior models and automated analysis tools (such as FDR). Now that we have more experience with architectural descriptions and abstractions through the development of WRIGHT, it is worth asking whether there might be ways that an underlying general formalism could be developed to better support architectural analysis. For example, we needed to add the distinction between event initiation and observation to CSP. Also, many of the nice algebraic properties of CSP are not automatically preserved by WRIGHT descriptions. For example, a configuration of components can not always be treated as equivalent to an atomically described component. (We discussed one difficulty, the initiator commits rule, in chapter 9.) The corresponding property of a process in CSP is that any composition of processes is another process. It is possible that an alternative basis might simplify WRIGHT's models, improve our ability to use architecture to structure reasoning, and provide a more convenient basis for combining different architectural models.

- **Dynamism:** A semantic model of particular interest would be one that would support the description and analysis of dynamic architectures. Finding an appropriate formal basis for dynamic architectures remains an open and active research topic. Some progress has been made by others. Rapide allows the dynamic creation of components, much as traditional object-oriented system can dynamically create new instances of objects. Darwin [MDEK95] uses the Pi Calculus [MPW92] as its basis, a formalism specifically designed to handle mobile processes. Architectural dynamism has also been modelled using the Chemical Abstract Machine [IW95] and graph grammars [LeM96] as the underlying formalism.

  These efforts, however, have not provided the kind of descriptive and analytic support for the non-dynamic aspects of architecture that WRIGHT has. We need to explore ways to combine characterizations of dynamism with the description and analysis of other aspects of a system. One possibility is to model a system's re-configurations as a sequence of distinct WRIGHT configurations, together with mappings between the configurations. These mappings would, for example, relate the computational state of components before and after the change.

- **Time:** A class of property that is often of critical concern for a software system is that of time. That is, there are many systems for which it is necessary to know not just *what* will occur in what order, as is modelled in CSP, but to determine *when* things will occur. There are existing extensions to CSP to handle time [RR86] that might be applied by substituting those models for CSP in WRIGHT. However, it is unclear what the effect of this change would have on WRIGHT's semantics (including, for example, how the consistency checks would need to be modified). Thus, research is needed to explore this kind of change to WRIGHT.

- **Architectural Refinement:** One especially important way of relating semantic models of a system is through refinement. Consider two descriptions of a system, one "abstract" and another "concrete." The abstract specification might simplify analysis of some aspects of a system while leaving other aspects unconstrained, and the concrete specification provide more details. If the concrete model guarantees all of the properties of the abstract model, and perhaps more, then we say that the concrete representation is a refinement of the abstract representation. An example of this might be the relation between a WRIGHT specification and an implementation of the same system. Clearly, the WRIGHT specification is more abstract than the implementation. At the same time, we want to show that there is a correspondence between the two such that any property that we prove about the WRIGHT description must also be true of the implementation. In formal specification, this task of relating an abstract specification to a more concrete one is that of finding a valid refinement mapping.

  While CSP does provide a compositional model of refinement (through the $\sqsubseteq$ relation), it is not clear that this is an adequate basis for architectural refinement (see, for example [Gar96] for a discussion of some of the complexities). It is certainly not an adequate basis when one of the descriptions is not based on CSP.

10.3.2.2 Architecture in the broader context

In this dissertation we have focussed on how software architecture, through WRIGHT, provides analytic and descriptive benefit to the software developer. In our evaluation (chapter 9) we discussed the cost of using WRIGHT, as that can impact its effectiveness in the context of engineering practice. However, another question is the appropriate use of architectural analysis in a development method, and how it can be effectively used in a larger software development context. Significant questions remain, including:

- **Architectural Selection:** How does the architect select an architectural style that is appropriate for a given situation? While WRIGHT provides models so that an architect can understand certain properties of styles, something more is needed to help the architect manage the tradeoffs between different styles. One style might provide the right properties but no guidance on constructing a system, or it might lack support at the implementation level. Another style might provide excellent support for implementation and make it easy to construct systems in the style, but provide less support for analysis. Shaw and Clements have carried out some preliminary work on classifying and selecting architectural styles [SC96, SC]. They argue that effective architectural selection depends on two factors, *careful discrimination* among candidates and *design guidance* on how to make appropriate choices. WRIGHT provides a starting point for the first factor, discrimination among styles, by allowing the architect to determine many properties of systems that can be constructed in a style. However, it remains to extend this discrimination to other properties and to provide design guidance.

- **Heterogeneous Systems:** What happens when a system doesn't match any given style, but instead, some parts match one style, while other parts match another style? WRIGHT has assumed that a given style description either applies, in its entirety, to a complete configuration, or it doesn't. The analytic leverage of a WRIGHT style is only available to configurations that can be completely characterized within that style. There has been some work on composing architectural configurations based on a shared set of architectural attributes and styles [AA96]. Future research must address the question of how, in general, a constraint can be bounded to apply to only part of a configuration and at the same time provide information about the configuration as a whole. One possible approach is to provide parameters to WRIGHT styles that would define the subset of the full system to which the style's predicates apply. While this would provide a means of applying styles to only part of a system, significant issues remain, including proof models for partially constrained configurations and consistency checks for combining styles in a single configuration with some overlap in scope.

- **Constructive Models of Style:** How can we use a declarative model of a style to a derive a constructive model of a style? As we discussed in chapter 9, WRIGHT's model of style is declarative, in that it defines what systems are in a style. Other current models of style are constructive, in that they provide mechanisms to construct an instance of a style. We argued that a declarative model of style provides increased benefits for description and analysis of systems. However, it does require that the architect construct a system before the style can be applied. An important question for future research is how construction guidance for architects can be derived from a style declaration.

- **Balancing Formalism and Informalism:** When is it appropriate to formalize an architecture, and when is an informal approach more appropriate? In this dissertation we have shown how architectural configurations and styles can be formalized, and how this provides both descriptive and analytic leverage. We must now ask ourselves: What are the advantages and disadvantages of a formal characterization relative to an informal one? When should one or the other be used? Up to now, there has been no practical, systematic means of taking a formal approach to software architecture. Now, with WRIGHT, formalization of architecture is a choice. As we gain more experience in the use of WRIGHT, we can begin to understand both the benefits that can be gained and an empirical model of the costs associated with formalization efforts. That will enable us to explore the question of when formalization is valuable relative to a more traditional informal effort.

  At the same time, we would argue that *both* kinds of characterization are useful, and that we should not necessarily view one or the other as better for architectural design. Rather, a complete exploration of a system or style should include both formal *and* informal characterizations. The benefits of the two kinds of characterization are complementary:

Informal	**Formal**
• Easy to understand	• Precise
• Shows how to build one	• Provable properties
• Structures design	• Structures analysis

As we can see from this list, a complete, fully-realized system of software architecture should seek to achieve all of these goals. Both formal models and informal understanding of software architecture are important to a successful engineering discipline, and we must explore ways in which the two can be combined to create a complete, realistic, and useful picture of a system's design.

10.3.2.3 Tailored formal notations

This thesis has shown how, in a particular domain (software architecture), by developing notations that directly reflect the abstractions of that domain, formal description and analysis can be made practical and effective. We believe that this approach of developing special models for particular domains is a fruitful one, and that it can and should be applied to other parts of the development process, such as requirements analysis and detailed design. It is unlikely, however, that WRIGHT itself will apply, because the abstractions that are appropriate for architecture may not match those that are appropriate for, say, requirements analysis. In that domain, for example, it is critical to distinguish between the existing condition of a system's environment and the desired condition while *not* specifying how the system will be constructed [Jac95b]. WRIGHT's success in formalization of architecture was achieved by narrowing the focus of the effort, making explicit the abstractions of the domain, and providing techniques for balancing formal effort against depth of analysis. Perhaps these techniques can be applied to other arenas as well.

10.3.2.4 Point-of-view in software development

Much of the power of the connector abstraction in WRIGHT comes from the fact that it requires the description be divided in terms of different points of view. That is, each of the participants in the interaction is described separately from the description of the overall effect of the interaction. Many of the inconsistencies that we have found through WRIGHT have been traceable to the failure to distinguish between what a given component *knows* and what is *globally true*. In the HLA case study, for example, it was stated as a requirement that the RTI connector could only be created once, but there was no mechanism provided for any component to find out whether the connector had already been created. Thus, no individual component could know enough to ensure the proper global behavior.

This point-of-view problem has also arisen elsewhere in the author's experience [GAO95]. One example is when a component is implemented on the basis of assumptions about the configuration in which it will occur. In effect, two points of view, that of a component and of the configuration as a whole, have been mixed. As a result, components that are supposed to be generally reusable are not, or can be reused only at the cost of considerable modification to the component or to the system in which it is to be deployed.

Lack of genericity in components is a major barrier to the goal of effective reuse and "component-level" programming, where architectural designs can be directly translated to implementations by the methodical use of existing infrastructure and componentry. One current research effort that attempts to decompose a specification in terms of different points of view is "aspect-oriented programming" [K^+]. The goal of that work is to permit the programmer to decompose a program specification in terms of semi-orthogonal issues of concern (*e.g.*, algorithm, distribution, data structure). The different specifications are then automatically combined into a single implementation.

We believe that exploring the role of point-of-view in specification, and in particular in architectural specification, has the potential to significantly improve our ability to systematically and effectively develop large-scale software systems. WRIGHT suggests several point-of-view distinctions (*e.g.*, between global and local, between a family of systems and a single instance) that provide leverage to specifiers and analysts. It is worth exploring whether there is some way that these distinctions can be systematized or otherwise more generally exploited in software development.

10.4 Epilogue: The Software Architecture Question

Throughout this dissertation, we have taken as a premise that providing an explicitly architectural approach to software development is the right thing to do. As we discussed earlier, there is a considerable body of effort in the development of systems through this approach, considering systems at a high level of abstraction as a composition of independent elements combined via independently characterized patterns of interaction.

Since its emergence in the late 1980's as an explicitly identified area of software engineering, software architecture has held out considerable promise for those working toward the goal of a true engineering discipline for software. By proposing a notation that supports the abstract modelling of systems that both exposes properties for analysis and ties directly to the implementation structure of systems, by promoting independence of reusable elements, by increasing the flexibility to compose those elements, by focussing on evolution of systems and on the reasoned development of multiple similar systems, the field of software architecture seems to present an exciting and promising direction in software development.

However, many of these benefits have yet to be realized. While there have been a number of significant success stories, both in the development of tools to support software development and in improved productivity through incorporation of architecture in the software process, the most expansive vision of a unifying conceptualization of architectural design that supports improved models of software and processes for software development remains only a future hope.

Nor is it possible to realize, or even test, such a vision unless there are rigorous, defined semantic foundations for the models and descriptions that people claim as the product of an architectural effort. It is only through the pursuit of such a foundation for software architecture that we can explore the possibilities of the area and begin to discover the strengths and weaknesses of the approach.

By itself, WRIGHT will not be able to achieve the full promise of architecture, nor to demonstrate that this promise is not achievable, but it does provide a more solid basis on which to explore the abstractions that define software architectures and architectural styles. In this respect, this work stands as a significant first step toward those larger goals.

Appendix A

Semantics of CSP

In this appendix, we describe the failures model of CSP, as well as provide some definitions of functions over CSP processes that we use in our definitions in previous chapters.

A.1 A Model of CSP: Alphabets, Traces, and Refusals

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 seen in this dissertation 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 finite sequence of events, and each refusal is a set of events. Thus, we write: $Failures(\mathsf{P}) \subseteq \mathsf{seq}\,\alpha\mathsf{P} \times \mathbb{P}\,\alpha\mathsf{P}$.

The traces of a process are those sequences of events that are permitted by the process. The process $\mathsf{P} = \mathsf{a} \rightarrow \mathsf{P} \,\square\, \mathsf{b} \rightarrow \mathsf{P}$, 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 $Traces(\mathsf{P})$.

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 and external 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 occuring (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 external choice, on the other hand, it cannot refuse any of the alternatives, 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 will typically be more than one refusal for a given trace. This occurs because refusals are *subset closed*, meaning that if a component can refuse a set of events, it can also refuse any subset of those events. Also, 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 $(\langle\rangle, \{\})$, $(\langle\rangle, \{\mathsf{a}\})$, 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.[1]

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:

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

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 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 $\alpha_i\mathsf{P}$, and the subset that is observed is $\alpha_o\mathsf{P}$.[2] This distinction has no effect on the behavior of the process, however.

A.2 Refinement

When we describe 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

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

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

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, by observing P, be able to detect that the process is P and not Q.[3] 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 7 (Refinement) *A process* $\mathsf{P} = (\alpha\mathsf{P}, F_{\mathsf{P}}, D_{\mathsf{P}})$ *is a refinement of* $\mathsf{Q} = (\alpha\mathsf{Q}, F_{\mathsf{Q}}, D_{\mathsf{Q}})$*, written* $\mathsf{Q} \sqsubseteq \mathsf{P}$*, if* $\alpha\mathsf{P} = \alpha\mathsf{Q}$ *and* $F_{\mathsf{P}} \subseteq F_{\mathsf{Q}}$ *and* $D_{\mathsf{P}} \subseteq D_{\mathsf{Q}}$.

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. $\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \sqcap \mathsf{f} \rightarrow \mathsf{Q} \sqsubseteq \mathsf{P} = \mathsf{e} \rightarrow \mathsf{P} \mathbin{[]} \mathsf{f} \rightarrow \mathsf{P}$

3. $\mathsf{Q} = \mathsf{e} \rightarrow \mathsf{Q} \mathbin{[]} \mathsf{f} \rightarrow \mathsf{Q} \not\sqsubseteq \mathsf{P} = \mathsf{e} \rightarrow \mathsf{P}$

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. That is,

$$\begin{array}{rl} & \mathit{Failures}(\mathsf{P}) = \{(\langle\rangle, \{\}), (\langle\rangle, \{f\}), (\langle e\rangle, \{f\}), (\langle e\rangle, \{\}), (\langle ee\rangle, \{f\}), ...\} \\ \subseteq & \mathit{Failures}(\mathsf{Q}) = \{(\langle\rangle, \{\}), (\langle\rangle, \{f\}), (\langle\rangle, \{e\}), (\langle e\rangle, \{f\}), (\langle e\rangle, \{e\}), (\langle f\rangle, \{e\}), ...\}. \end{array}$$

In example 2, process P does not change the traces of Q, but it removes f from the refusals, reducing non-determinism. In this case, $\mathit{Failures}(\mathsf{P}) = \{(\langle\rangle, \{\}), (\langle e\rangle, \{\}), (\langle f\rangle, \{\})...\}$. In example 3, P is *not* a refinement of Q, because it refuses f when Q does not. $(\langle\rangle, \{f\}) \in \mathit{Failures}(\mathsf{P})$ but $(\langle\rangle, \{f\}) \notin \mathit{Failures}(\mathsf{Q})$

A.3 Auxiliary Definitions

In our formalization of WRIGHT, we used a number of standard functions over CSP processes. These are defined in this section.

[3] We may, however, be able by observing Q to detect that it is not P.

A.3.1 Renaming

A process is renamed by applying a function from events to events. This function must be injective, so that undesired aliasing does not occur, and it must not rename $\surd$, so that termination is properly modelled.

Definition 8 (Renaming) *If f is an injective function from events to events, such that $\surd$ is not in the domain of f, and* $\mathsf{P} = (A, F, D)$ *is a CSP process, then* $f(\mathsf{P}) = (A', F', D')$ *where*

$$\begin{aligned} A' &= f(\!|A|\!) \\ F' &= \{(t', r') \mid \exists (t, r) \in F \mid t' = f \circ t \wedge r' = f(\!|r|\!)\} \\ D' &= \{d' : \operatorname{seq} A' \mid (\exists\, d : D \mid d' = f \circ d)\}. \end{aligned}$$

In this definition, $f \circ g$ is the composition of functions f and g. $f(\!|S|\!)$ is the relational image of the set S under f.

Relabelling is a simple case of renaming:

Definition 9 (Relabelling) *For a process* P *and a name* L, L:P = f_L(P), *where* f_L(e) = L.e *for all events* e $\neq \surd$.

A.3.2 Projection

Projection is actually defined in terms of hiding of events. We hide events as follows:

Definition 10 *For any process* $\mathsf{P} = (A, F, D)$ *and event set E,* $\mathsf{P} \setminus E = (A - E, F', D')$ *where*

$$\begin{array}{lllll} F' = & & \{(t \restriction (A - E), X) & \mid & (t, X \cup E) \in F\} \\ & \cup & \{(t, X) & \mid & t \in D'\} \\ D' = & & \{(s \frown t) \restriction (A - E) & \mid & \quad s \in D \\ & & & & \vee\ \forall\, n \bullet \exists\, u : E^* \bullet \#u > n \wedge (s \frown u, \varnothing) \in F\}. \end{array}$$

In this definition, a trace projection ($t \restriction E$) 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 $\langle acadbcbc \rangle \restriction \{a, b\} = \langle aabb \rangle$.

Process projection is simply the opposite of hiding:

Definition 11 *For any process* P *and event set E,* $\mathsf{P} \restriction E = \mathsf{P} \setminus (\alpha\mathsf{P} - E)$.

A.4 A Note Regarding § and the ';' Operator

Our use of CSP in WRIGHT differs from other interpretations in one important respect: We use the process § and, hence the $\surd$ event to represent a *willingness* to terminate rather than a *decision* to terminate. In particular, our interpretation of the expression "P ⊓ §" is slightly different from other interpretations, such as that proposed by Roscoe [Ros95] and used in the implementation of FDR [FDR93].

A.4.1 Non-standard Interpretation

The difficulty with the interpretation of sequencing arises because of an omission in the original formulation of CSP, as described by Hoare in [Hoa85]. That treatment does not define the meaning of "$\mathsf{P} \square §$" and, in fact, explicitly prohibits it [Hoa85, p. 178].

The basic problem is that we need to understand the meaning of this construct as it influences other CSP operators. The operator that obviously is most affected is sequencing. The question is: how do we interpret the following process?

$$(\mathsf{P} \square §); \mathsf{Q}$$

According to some interpretations of sequencing, such as that found in [Ros95], this process is equivalent to

$$(\mathsf{P}; \mathsf{Q}) \sqcap \mathsf{Q}.$$

The problem with this definition is that it introduces non-determinism when combining deterministic processes. Although this is always possible (due to the way external choice models aternatives between the same event), it seems desirable to minimize the extent to which this occurs. I would argue that we want it instead to be equivalent to

$$(\mathsf{P}; \mathsf{Q}) \square \mathsf{Q}.$$

For example, suppose we are specifying a collection of possible interface specifications such as those described in chapter 4. The core of the protocol is a pair of actions, request then result. There is also a surrounding initialization and clean-up phase, just as we saw in the AEGIS system. It would be desirable to build the protocol variants up out of simple parts, as follows:

OneRequest = request→result→§
ManyRequests = (OneRequest ; ManyRequests) $\square$ §
WrapperMany = init →ManyRequests ; close→§
WrapperOne = init →OneRequest ; close→§

WrapperMany represents the case where many request, result pairs are permitted before the close event, while WrapperOne represents the case where only one such pair is permitted. What we would like is to have:

WrapperMany = init →ContMany
ContMany = request →result →ContMany
        $\square$ close→§
WrapperOne = init →request→result→close→§

which describes the desired protocols, only without the structuring that exposes their relationship. In order to achieve this effect, we must revise the definition of "P;Q" so that it treats "(P $\square$ §); Q" as a deterministic choice between P and Q. We therefore use the following alternative definition of ';':

Definition 12 (Sequence) *If* P = (A, F, D) *and* Q = (A, F', D'), *then* P;Q = (A, F'', D'') *s.t.*

$$\begin{array}{rlll} D'' = & & \{s & \mid s \in D \wedge \langle\surd\rangle \notin s\} \\ & \cup & \{s \frown t & \mid (s \frown \langle\surd\rangle, \varnothing) \in F \wedge \langle\surd\rangle \notin s \wedge t \in D'\} \\ \\ F'' = & & \{(s, X) & \mid (s, X \cup \{\surd\}) \in F) \wedge \langle\surd\rangle \notin s\} \\ & \cup & \{(s, X) & \mid (s \frown \langle\surd\rangle, \varnothing) \in F \wedge (s, X - \{\surd\}) \in F \wedge (\langle\rangle, X) \in F'\} \\ & \cup & \{(s \frown t, X) & \mid (s \frown \langle\surd\rangle, \varnothing) \in F \wedge (t, X) \in F' \wedge t \neq \langle\rangle\} \\ & \cup & \{(s, X) & \mid s \in D''\} \end{array}$$

Compare this to Hoare's definition [Hoa85, p. 179]:

$$\begin{array}{rlll} D'' = & & \{s & \mid s \in D \wedge \langle\surd\rangle \notin s\} \\ & \cup & \{s \frown t & \mid (s \frown \langle\surd\rangle, \varnothing) \in F \wedge \langle\surd\rangle \notin s \wedge t \in D'\} \\ \\ F'' = & & \{(s, X) & \mid (s, X \cup \{\surd\}) \in F) \wedge \langle\surd\rangle \notin s\} \\ & \cup & \{(s \frown t, X) & \mid (s \frown \langle\surd\rangle, \varnothing) \in F \wedge (t, X) \in F'\} \\ & \cup & \{(s, X) & \mid s \in D''\} \end{array}$$

Our definition is the same as Hoare's definition except in how it handles traces s where $s \frown \langle\surd\rangle \in traces(\mathsf{P})$ and $\surd$ is not in the refusals of P after trace s. In this case, we require that *both* P and Q refuse an event before it can be refused by the process.[4] Thus, we interpret termination as a *deterministic* rather than a *non-deterministic* action by a process.

A.4.2 Justification of Interpretation

In order to use this interpretation of the sequencing operator, we must ensure that it is consistent with the rest of CSP and that it has certain desirable properties. There are three steps to defining an operator for CSP [Ros95].

A.4.2.1 Operator is Total

First, we must show that the operator is *total* over well-defined processes. That is, we must show that whenever P and Q are processes, P;Q is also a process. The following laws define a well-formed process [Hoa85, p. 130]:
For a process P $= (A, F, D)$

C0: $(\langle\rangle, \{\}) \in F$
C1: $(s \frown t, X) \in F \Rightarrow (s, \{\}) \in F$
C2: $(s, Y) \in F \wedge X \subseteq Y \Rightarrow (s, X) \in F$
C3: $(s, X) \in F \wedge x \in A \Rightarrow (s, X \cup \{x\}) \in F \vee (s \frown \langle x\rangle, \{\}) \in F$
C4: $D \subseteq domain(F)$
C5: $s \in D \wedge t \in A^* \Rightarrow s \frown t \in D$
C6: $s \in D \wedge X \subseteq A \Rightarrow (s, X) \in F$

All of these laws hold for our definition, and thus the operator is well-defined.

[4] In our definition, if $\surd$ is in the refusals then the first line of the failures definition dominates, and P alone may refuse events.

A.4.2.2 Continuity and Monotonicity

The second step in justifying a process is to show that the operator is continuous. This ensures that the operator is well-defined in recursive definitions (because of the existence of fixed-points) and that it is monotone with respect to recursion. That is, for any processes P, P', Q, such that $\mathsf{P} \sqsubseteq \mathsf{P}'$, $\mathsf{P};\ \mathsf{Q} \sqsubseteq \mathsf{P}';\ \mathsf{Q}$ and $\mathsf{Q};\ \mathsf{P} \sqsubseteq \mathsf{Q};\ \mathsf{P}'$. Together, the properties of continuity and monotonicity ensure that all algebraic laws about processes that do not refer to the operator in question are guaranteed still to hold.

In order to show that our definition is continuous, we must show that:

$$(\mathsf{P};\ \textstyle\bigsqcup_{i \geq 0} \mathsf{Q}_i) = \textstyle\bigsqcup_{i \geq 0}(\mathsf{P};\ \mathsf{Q}_i)$$

and

$$(\textstyle\bigsqcup_{i \geq 0} \mathsf{Q}_i);\ \mathsf{P} = \textstyle\bigsqcup_{i \geq 0}(\mathsf{Q}_i;\ \mathsf{P})$$

for any chain of Q_i s.t.

$$\forall\, i, j \mid i < j \bullet \mathsf{Q}_i \sqsubseteq \mathsf{Q}_j$$

For a process, the limit $\bigsqcup$ is defined as:

$$\textstyle\bigsqcup_{n \geq 0}(A, F_n, D_n) = (A, \bigcap_{n \geq 0} F_n, \bigcap_{n \geq 0} D_n)$$

Because we depend on only a single trace or failure for each part of our definition, it is easy to see that our operator is continuous. For example, consider one of the conditions that we have changed, the case where a trace of P could terminate but does not (necessarily) refuse $\checkmark$. Here, $\mathit{Failures}(\mathsf{P}_n) = F_n$, and so $\mathit{Failures}(\bigsqcup_{n \geq 0} \mathsf{P}_i) = \bigcap_{n \geq 0} F_n$.

$$\begin{array}{lll}
 & & (s, X) \in \{\{(s, X) \mid\ (s \frown \langle \checkmark \rangle, \varnothing) \in \bigcap_{n \geq 0} F_n \\
 & & \qquad\qquad\qquad \wedge\ (s, X - \{\checkmark\}) \in \bigcap_{n \geq 0} F_n \\
 & & \qquad\qquad\qquad \wedge\ (\langle\rangle, X) \in F'\} \\
\Leftrightarrow & \forall\, n \geq 0 \bullet & (s, X) \in \{\{(s, X) \mid\ (s \frown \langle \checkmark \rangle, \varnothing) \in F_n \\
 & & \qquad\qquad\qquad \wedge\ (s, X - \{\checkmark\}) \in F_n \\
 & & \qquad\qquad\qquad \wedge\ (\langle\rangle, X) \in F'\} \\
\Leftrightarrow & & (s, X) \in \bigcap_{n \geq 0}\{\{(s, X) \mid\ (s \frown \langle \checkmark \rangle, \varnothing) \in F_n) \\
 & & \qquad\qquad\qquad\qquad \wedge\ (s, X - \{\checkmark\}) \in F_n) \\
 & & \qquad\qquad\qquad\qquad \wedge\ (\langle\rangle, X) \in F'\} \\
\Rightarrow & & (s, X) \in \mathit{Failures}(\bigsqcup(\mathsf{P}_i;\ \mathsf{Q}))
\end{array}$$

The other cases and directions proceed similarly.

A.4.2.3 Laws

The third step in defining a new operator is to define and prove algebraic laws involving the operator. This ensures that the operator has any desired properties and that one can reason effectively about processes defined using the operators.

Common Laws

Roscoe [Ros95] and Hoare [Hoa85] define the following laws for ';':

$$\begin{array}{rcl}
(\mathsf{P} \sqcap \mathsf{Q}); \mathsf{R} & = & (\mathsf{P}; \mathsf{R}) \sqcap (\mathsf{Q}; \mathsf{R}) \\
\mathsf{P}; (\mathsf{Q} \sqcap \mathsf{R}) & = & (\mathsf{P}; \mathsf{Q}) \sqcap (\mathsf{P}; \mathsf{R}) \\
\mathsf{P}; (\mathsf{Q}; \mathsf{R}) & = & (\mathsf{P}; \mathsf{Q}); \mathsf{R} \\
\S; \mathsf{P} & = & \mathsf{P} \\
\mathsf{P}; \S & = & \mathsf{P}
\end{array}$$

$$\begin{array}{rcll}
(\mathsf{a} \rightarrow \mathsf{P}); \mathsf{Q} & = & \mathsf{a} \rightarrow (\mathsf{P}; \mathsf{Q}) & \quad (\text{for } \mathsf{a} \neq \surd) \\
\mathbf{STOP}; \mathsf{Q} & = & \mathbf{STOP} &
\end{array}$$

These state that $\sqcap$ distributes through sequencing, that sequencing is associative, that $\S$ is a left and right unit of sequencing, that prefix and sequencing are associative, and that **STOP** is a left zero of sequencing.

All of these laws hold for our operator definition. Here, we will show only an example proof, that $(\mathsf{P} \sqcap \mathsf{Q}); \mathsf{R} = (\mathsf{P}; \mathsf{R}) \sqcap (\mathsf{Q}; \mathsf{R})$. We will further show only that the failures are equivalent, since we did not change the definition of divergences.

Let $F_P = failures(\mathsf{P}), F_Q = failures(\mathsf{Q}), F_R = failures(\mathsf{R}), F_{PQ} = failures(\mathsf{P} \sqcap \mathsf{Q}), F_{PR} = failures(\mathsf{P};\ \mathsf{R}), F_{QR} = failures(\mathsf{Q};\ \mathsf{R})$. Further, let $F = failures((\mathsf{P} \sqcap \mathsf{Q});\ \mathsf{R})$, $F' = failures((\mathsf{P};\ \mathsf{R}) \sqcap (\mathsf{Q};\ \mathsf{R}))$. Then:

$$\begin{array}{rll}
 & (s \frown t, X) \in F & \\
\Leftrightarrow & (s, X \cup \{\surd\}) \in F_{PQ}) \wedge \langle \surd \rangle \notin s \wedge t = \langle\rangle & (\text{A.1}) \\
\vee & (s \frown \langle \surd \rangle, \varnothing) \in F_{PQ} \wedge (s, X - \{\surd\}) \in F_{PQ} & \\
 & \wedge (\langle\rangle, X) \in F_R \wedge t = \langle\rangle & (\text{A.2}) \\
\vee & (s \frown \langle \surd \rangle, \varnothing) \in F_{PQ} \wedge (t, X) \in F_R \wedge t \neq \langle\rangle & (\text{A.3}) \\
\vee & s \in D & (\text{A.4})
\end{array}$$

$$\begin{array}{rrll}
(\text{A.1}) & \Rightarrow & ((s, X \cup \{\surd\}) \in F_P \wedge \langle \surd \rangle \notin s \wedge t = \langle\rangle) & (\text{A.5}) \\
 & & \vee ((s, X \cup \{\surd\}) \in F_Q \wedge \langle \surd \rangle \notin s \wedge t = \langle\rangle) & (\text{A.6}) \\
(\text{A.2}) & \Rightarrow & ((s \frown \langle \surd \rangle, \varnothing) \in F_P \vee (s \frown \langle \surd \rangle, \varnothing) \in F_Q) & \\
 & & \wedge ((s, X - \{\surd\}) \in F_P \vee (s, X - \{\surd\}) \in F_Q) & \\
 & & \wedge (\langle\rangle, X) \in F_R \wedge t = \langle\rangle & \\
 & \Rightarrow & ((s \frown \langle \surd \rangle, \varnothing) \in F_P \wedge (s, X - \{\surd\}) \in F_P & \\
 & & \quad \wedge (\langle\rangle, X) \in F_R \wedge t = \langle\rangle) & (\text{A.7}) \\
 & & \vee ((s \frown \langle \surd \rangle, \varnothing) \in F_Q \wedge (s, X - \{\surd\}) \in F_Q & \\
 & & \quad \wedge (\langle\rangle, X) \in F_R \wedge t = \langle\rangle) & (\text{A.8}) \\
(\text{A.3}) & \Rightarrow & ((s \frown \langle \surd \rangle, \varnothing) \in F_P \wedge (t, X) \in F_R \wedge t \neq \langle\rangle) & (\text{A.9}) \\
 & & \vee ((s \frown \langle \surd \rangle, \varnothing) \in F_Q \wedge (t, X) \in F_R \wedge t \neq \langle\rangle) & (\text{A.10})
\end{array}$$

These result in the solution, since (recalling that $failures(P \sqcap Q) = failures(P) \cup failures(Q)$):

$$\begin{aligned}
\text{(A.5)} \vee \text{(A.7)} \vee \text{(A.9)} &\Rightarrow (s, X) \in F_{PR} \subseteq F' \\
\text{(A.6)} \vee \text{(A.8)} \vee \text{(A.10)} &\Rightarrow (s, X) \in F_{QR} \subseteq F' \\
\text{(A.4)} &\Rightarrow (s, X) \in F'
\end{aligned}$$

and thus $F \subseteq F'$. The converse, that $F' \subseteq F$ proceeds similarly, as do the proofs of the other laws.

Laws that Change

But what laws are changed by our interpretation of $\surd$ as a shared rather than a hidden event? There is one difference, and it is in how the consistent use of $\surd$ event is enforced. Roscoe treats $\surd$ as an event that is initiated, and thus requires the following:

$$s \frown \langle \surd \rangle \in traces(P) \Rightarrow (s, \alpha P - \{\surd\}) \in failures(P)$$

This basically states that whenever a process *can* terminate, it can always refuse to do anything else. In Hoare's less general treatment of termination, he appears to have a similar interpretation, using the following law:

$$s \frown \langle \surd \rangle \in traces(P) \Rightarrow P/s \sqsubseteq \S$$

We rather view termination as a global behavior, and therefore weaken this condition. We would rather indicate that if a process *does* terminate, then it does not do anything afterwards:

$$s \in traces(P) \wedge \langle \surd \rangle \in s \Rightarrow \exists\, s' \mid s = s' \frown \langle \surd \rangle \wedge \langle \surd \rangle \notin s'$$

In Hoare's notation, this would be stated:

$$s \frown \langle \surd \rangle \in traces(P) \Rightarrow P/(s \frown \langle \surd \rangle) = STOP$$

This law holds for all processes that introduce $\surd$ using only the $\S$ process, and ensures that termination is properly modeled.

In addition to these laws, there is one more that holds for our definition and not for the alternative, non-deterministic interpretation of sequence:

$$(\mathsf{P} \Box \mathsf{Q}); \mathsf{R} = (\mathsf{P}; \mathsf{R}) \Box (\mathsf{Q}; \mathsf{R})$$

This states that the $\Box$ operator distributes on the left-hand side of sequencing. This does not hold if we interpret $\surd$ as a hidden event, rather than a shared event. The right-hand version of this law does not hold for either interpretation:

$$\mathsf{P}; (\mathsf{Q} \Box \mathsf{R}) \neq (\mathsf{P}; \mathsf{Q}) \Box (\mathsf{P}; \mathsf{R})$$

This is because moving P through the $\Box$ results in the repetition of the initial events in P on both sides of the operator and hence a non-deterministic process.

A.4.3 Possible Disadvantage to Interpretation

Given that this alternative interpretation of $\surd$ and sequencing obeys at least some form of all of the laws of the other interpretation as well as at least one useful additional law, one might ask why the other interpretation would ever be used. The answer lies in the problem of automating analysis. In constructing tools such as FDR, it is not the algebraic laws that determine how difficult it is to analyze a system, but rather the size of the state-graph representation and the complexity of generating one process' state-graph from another. Here, the non-deterministic interpretation wins, because it is much easier to manipulate the state-graph representations for the non-deterministic interpretation than for the deterministic interpretation. This is because in the non-deterministic version, one can generate the graph "P;Q" from the graphs P and Q simply by moving all $\surd$ transition edges in P to a τ (hidden) transition edge to the start state of graph Q. Calculating the new state-graph for our interpretation of "P;Q" is much more complex, and can result in the need to generate new transitions from states of P to arbitrary states of Q, rather than just to the start state. In effect, the start state of Q has to be *overlaid* with the states of P, rather than simply referenced from those states. Thus, the use of our interpretation makes it more difficult to construct an exhaustive checker. Also, because it increases the number of transitions, it can, in situations that were near the limit of the checker's capacity, cause automated checking to become impractical. It can not, however, result in an infinite number of states when the other interpretation does not.

Appendix B

Details of WRIGHT Specification of HLA

This appendix provides details of the WRIGHT description of HLA. Chapter 6 provides an overview of HLA and a discussion of the techniques used in this description.

B.1 Simulation Interface

B.1.1 Separate Parts of Interface

There are five sections to the RTI interface, Federation Management, Declaration Management, Object Management, Ownership Management, and Time Management. Of these, this specification ignores time management. The reason for this is that it is a complex issue that is orthogonal to the others. That is, the events involved with time management do not, for the most part, affect the pattern of the other events. Of course, Federation Management events affect the time management events, but not the other way around.

We have also chosen mostly to abstract away from the parameters passed by the messages. This is again because the patterns of data are complicated and, for the most part, the control patterns can be described without reference to the actual data values. When we need to, such as when success or failure of a request matters, we have created a pair of events to represent the two alternatives. For example, attrOwnAcquisitionNotifySecured and attrOwnAcquisitionNotifyRejected represent two possible responses that are conveyed by the same message in the actual specification.

The formal WRIGHT specification of the RTI has two main parts: the SimInterface interface type, and the RTI connector. The SimInterface is used as the role of the connector, and represents the protocol for interaction with the simulation *from the point of view of a single federate*. The combined behavior of several federates interacting is specified in the RTI's **Glue**.

We begin by specifying the SimInterface. This specification is structured, to the extent possible, to match the management categories that are described in the IFSpec. Thus, there are four main processes that are combined in SimInterface: FedMgmt, DeclMgmt, ObjMgmt, and OwnMgmt. Each has an independent alphabet and represents the constraints that can be represented for each management group independently.

In addition to the constraints on each individual group, there are rules that cross group boundaries. For example, although the pause controlling events are located under Federation

Management, there are limits on all events during a pause. For example, the federate need not be prepared to receive an object attribute update after a pause has been achieved.

Each of these "cross group" constraints is represented as a separate process that is part of the total SimInterface. The reason for this is so that it is as clear as possible how they correspond to the IFSpec document, either from the pre- and post-conditions of various events or from the more general discussions that occur.

Note that the "exceptions" mentioned in the IFSpec are not represented here. There are two reasons for this: first, the general goal of an interface is to *avoid* exceptional conditions. Therefore, to the extent possible, we described patterns of events that prevented exceptions rather than leading to them. Second, the status of exceptions as to whether they actually generated communication events was only clarified as a result of the exercise of formalizing the IFSpec in this WRIGHT specification. Therefore, this initial version made the assumption that exceptions were anomalous conditions (that would be represented by undesirable states or traces of events) rather than actual messages that are communicated from the RTI to a federate, or vice versa. As a result of our formalization effort, later drafts of the IFSpec indicate that an exception can generate message traffic, and so ought in some way to be included in the specification. In that case, it would be desirable to show that, due to the structure of the protocol, these exception events may never occur. This would be another reasonable way of using WRIGHT. We chose instead to have the protocols only describe desirable behaviors, and to detect as deadlocks situation where an undesirable behavior would have occurred.

The first "group" process that forms part of SimInterface is FedMgmt. Its alphabet is as follows:

$$
\begin{aligned}
\alpha\mathsf{FedMgmt} = \{\; & \overline{\mathsf{createFedExecution}}, \overline{\mathsf{destroyFedExecution}},\\
& \overline{\mathsf{joinFedExecution}}, \overline{\mathsf{resignFedExecution}},\\
& \overline{\mathsf{requestPause}}, \mathsf{schedulePause}, \overline{\mathsf{pauseAcheived}},\\
& \overline{\mathsf{requestResume}}, \mathsf{scheduleResume}, \overline{\mathsf{resumeAchieved}},\\
& \overline{\mathsf{scheduleFedSave}}, \mathsf{startFedSave},\\
& \overline{\mathsf{fedSaveBegun}}, \overline{\mathsf{fedSaveCompete}},\\
& \overline{\mathsf{requestRestore}}, \mathsf{restore}, \overline{\mathsf{restoreComplete}},\\
& \overline{\mathsf{submitQuery}}, \mathsf{query}, \mathsf{queryResult}\}
\end{aligned}
$$

The FedMgmt process is responsible for controlling events that lead to federation creation and destruction, the federate joining an resigning from the federation, queries about the state of the simulation (meta-queries), as well as pause/resume and checkpointing.

Initially, the federate may need to create the RTI execution (if it doesn't exist). After that, it must join the federation before it may do anything else (process StartFedMgmt). After that, it is in the "normal" operating condition (ContFedMgmt). It may select from a set of management events, or it may terminate its own execution (EndFedMgmt). Normal execution includes both sending events and receiving events (FedWaitForEvent).

The process FedPaused represents the same state as ContFedMgmt, except that in this state, the simulation is paused. Thus, resumes are expected rather than pauses, and the federation is not permitted to request additional pauses.

In both FedWaitForEvent and PauseWaitForEvent there is an additional message that is not in the IFSpec: noMessage. This event represents the case where the federate looks for an event and there is none waiting.

Process FedMgmt = ($\overline{\text{createFedExecution}}$ $\rightarrow$StartFedMgmt) $\sqcap$StartFedMgmt
where
StartFedMgmt = $\overline{\text{joinFedExecution}}$ $\rightarrow$ContFedMgmt
ContFedMgmt = EndFedMgmt
$\sqcap$ $\overline{\text{requestPause}}$ $\rightarrow$ContFedMgmt
$\sqcap$ $\overline{\text{submitQuery}}$ $\rightarrow$queryResult $\rightarrow$ContFedMgmt
$\sqcap$ $\overline{\text{scheduleFedSave}}$ $\rightarrow$ContFedMgmt
$\sqcap$ $\overline{\text{requestRestore}}$ $\rightarrow$ContFedMgmt
$\sqcap$ $\overline{\text{submitQuery}}$ $\rightarrow$ContFedMgmt
$\sqcap$ WaitForEvent
WaitForEvent = schedulePause $\rightarrow$$\overline{\text{pauseAchieved}}$ $\rightarrow$FedPaused
$\square$ startFedSave $\rightarrow$DoFedSave; ContFedMgmt
$\square$ restore $\rightarrow$$\overline{\text{restoreComplete}}$ $\rightarrow$ContFedMgmt
$\square$ query $\rightarrow$ContFedMgmt
$\square$ queryResult $\rightarrow$ContFedMgmt
$\square$ noMessage $\rightarrow$ContFedMgmt
DoFedSave = $\overline{\text{fedSaveBegun}}$ $\rightarrow$$\overline{\text{fedSaveComplete}}$ $\rightarrow\S$
FedPaused = $\overline{\text{requestResume}}$ $\rightarrow$FedPaused
$\sqcap$ $\overline{\text{submitQuery}}$ $\rightarrow$queryResult $\rightarrow$ContFedMgmt
$\sqcap$ $\overline{\text{scheduleFedSave}}$ $\rightarrow$FedPaused
$\sqcap$ $\overline{\text{requestRestore}}$ $\rightarrow$FedPaused
$\sqcap$ $\overline{\text{submitQuery}}$ $\rightarrow$FedPaused
$\sqcap$ PauseWaitForEvent
PauseWaitForEvent = scheduleResume $\rightarrow$$\overline{\text{resumeAchieved}}$ $\rightarrow$ContFedMgmt
$\square$ startFedSave $\rightarrow$DoFedSave; FedPaused
$\square$ restore $\rightarrow$$\overline{\text{restoreComplete}}$ $\rightarrow$FedPaused
$\square$ query $\rightarrow$FedPaused
$\square$ queryResult $\rightarrow$FedPaused
$\square$ noMessage $\rightarrow$FedPaused
EndFedMgmt = $\overline{\text{resignFedExecution}}$ $\rightarrow$(($\overline{\text{destroyFedExecution}}$ $\rightarrow\S$) $\sqcap$ $\S$)

The second management group is DeclMgmt. Its alphabet is:

$$\begin{aligned}\alpha\text{DeclMgmt} = \{ \; & \overline{\text{publishObjClass}}, \overline{\text{publishObjAttr}}, \\ & \overline{\text{publishIntrClass}}, \overline{\text{subscribeIntrClass}}, \\ & \overline{\text{subscribeObjClass}}, \overline{\text{subscribeObjAttr}}, \\ & \text{controlUpdates}, \text{controlInteractions}\}\end{aligned}$$

DeclMgmt handles declaration management; namely, the controlling of what objects and attributes the federate is capable of publishing and interested in receiving information about, respectively. There is no mention of individual instances in this management group; that is handled by the next two, ownership and object management. There are no significant preconditions reflected in this group and this model, because what preconditions there are, are related to consistency of data with the federation object model. The object model is, obviously, ignored, since it affects parameter values rather than control state.

Process DeclMgmt = $\overline{\text{publishObjClass}} \rightarrow$DeclMgmt
$\sqcap$ $\overline{\text{publishObjAttr}} \rightarrow$DeclMgmt
$\sqcap$ $\overline{\text{publishIntrClass}} \rightarrow$DeclMgmt
$\sqcap$ $\overline{\text{subscribeIntrClass}} \rightarrow$DeclMgmt
$\sqcap$ $\overline{\text{subscribeObjClass}} \rightarrow$DeclMgmt
$\sqcap$ $\overline{\text{subscribeObjAttr}} \rightarrow$DeclMgmt
$\sqcap$ WaitForEvent
$\sqcap$ §
where
WaitForEvent = controlUpdates $\rightarrow$DeclMgmt
$\Box$ controlInteractions $\rightarrow$DeclMgmt
$\Box$ noMessage $\rightarrow$DeclMgmt

The Object management group covers communication about specific object/attribute instances. In this model, in order to maintain finite-state while approximating the preconditions, we treat the state of the federate as falling into two broad categories: One in which it is not possible to own objects and another in which it is possible. The difference in this model is whether the federate has object IDs allocated to it. This is not quite the correct distinction, since, for example, a federate could acquire ownership of an object (via the ownership management functions) that it did create. It is accurate with respect to the creation of objects, however. WaitForEvent does not need to distinguish the two states; it is, in effect, a macro that represents a single observed event. The use of a terminating process followed by ';' indicates that the continuation of the process is different based on the context in which it is used.

$$
\begin{aligned}
\alpha\text{ObjMgmt} = \{ \; & \overline{\text{idRequest}}, \\
& \overline{\text{instantiateObj}}, \overline{\text{deleteObj}}, \\
& \text{instantiateDiscoveredObj}, \text{removeObj}, \\
& \overline{\text{updateAttrValues}}, \text{reflectAttrValue}, \\
& \overline{\text{cancelObjReflection}}, \\
& \overline{\text{sendInteraction}}, \text{receiveInteraction}, \\
& \overline{\text{requestAttrValueUpdate}}, \text{provideAttrValueUpdate} \}
\end{aligned}
$$

Process ObjMgmt = $\overline{\text{idRequest}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{sendInteraction}} \rightarrow$ObjMgmt
$\sqcap$ $\overline{\text{requestAttrValueUpdate}} \rightarrow$ObjMgmt
$\sqcap$ WaitForEvent; ObjMgmt
$\sqcap$ $\overline{\text{cancelObjReflection}} \rightarrow$ObjMgmt
$\sqcap$ §
where
MayOwnObjs = $\overline{\text{idRequest}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{instantiateObj}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{sendInteraction}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{requestAttrValueUpdate}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{deleteObj}} \rightarrow$MayOwnObjs
$\sqcap$ $\overline{\text{updateAttrValues}} \rightarrow$DoesOwnObjs

$\sqcap$ WaitForEvent; MayOwnObjs
$\sqcap$ $\overline{\text{cancelObjReflection}}$ $\rightarrow$MayOwnObjs
$\sqcap$ §
WaitForEvent = instantiateDiscoveredObj $\rightarrow$§
$\square$ reflectAttrValue $\rightarrow$§
$\square$ receiveInteraction $\rightarrow$§
$\square$ provideAttrValueUpdate$\rightarrow$§
$\square$ noMessage $\rightarrow$§

The ownership management group consists of a number of sub-protocols regarding transfer of ownership of objects or attributes. Thus, it is one of the more complex groups. The distinction between the different sub-protocols is whether the change is initiated by the federate or some other federate, whether it is the entire object or only an attribute, and whether the federate is receiving or releasing ownership. Each has essentially the same initiation-response pattern.

$$\begin{aligned}\alpha\text{OwnMgmt} = \{ \; & \overline{\text{reqAttrOwnDivestiture}}, \text{attrOwnDivestitureNotifyRelease}, \\ & \text{attrOwnDivestitureNotifyRetain}, \overline{\text{uncondAttrOwnDivestiture}}, \\ & \text{reqAttrOwnAssumption}, \\ & \overline{\text{reqAttrOwnAcquisition}}, \text{attrOwnAcquisitionNotifySecured}, \\ & \text{attrOwnAcuisitionNotifyRejected}, \text{reqAttrOwnRelease}, \\ & \overline{\text{queryAttrOwnership}}, \\ & \overline{\text{reqDeletePrivAcquisition}}, \text{reqDeletePrivRelease}, \\ & \text{deletePrivNotifySecured}, \text{deletePrivNotifyRejected}\}\end{aligned}$$

Process OwnMgmt = $\overline{\text{reqAttrOwnDivestiture}}$ $\rightarrow$DivestResponse
$\sqcap$ $\overline{\text{uncondAttrOwnDivestiture}}$ $\rightarrow$OwnMgmt
$\sqcap$ $\overline{\text{reqAttrOwnAcquisition}}$ $\rightarrow$OwnAcquisitionResponse
$\sqcap$ $\overline{\text{queryAttrOwnership}}$ $\rightarrow$OwnMgmt
$\sqcap$ $\overline{\text{reqDeletePrivAcquisition}}$$\rightarrow$DeleteAcquisitionResponse
$\sqcap$ WaitForEvent
$\sqcap$ §
where
DivestResponse = attrOwnDivestitureNotifyRelease $\rightarrow$OwnMgmt
$\square$ attrOwnDivestitureNotifyRetain $\rightarrow$OwnMgmt
OwnAcquisitionResponse = attrOwnAcquisitionNotifySecured $\rightarrow$OwnMgmt
$\square$ attrOwnAcquisitionNotifyRejected $\rightarrow$OwnMgmt
DeleteAcquisitionResponse = deletePrivNotifySecured $\rightarrow$OwnMgmt
$\square$ deletePrivNotifyRejected $\rightarrow$OwnMgmt
WaitForEvent = reqAttrOwnAssumption $\rightarrow$(OwnAcquisitionResponse $\sqcap$ OwnMgmt)
$\square$ reqOwnAttrRelease $\rightarrow$OwnMgmt
$\square$ reqDeletePrivRelease $\rightarrow$OwnMgmt
$\square$ noMessage

B.1.2 Interface Type Specification

The process SimInterface will be the role specification for the RTI. It may also, therefore, be used as a port spec for a federate, but individual federates might prefer to have more refined interface specifications. For example, a "Federate (Manager)" might only use the federation management functions, while a non-manager might only use the join/resign part of the management functions.

$$\alpha\mathsf{SimInterface} = \alpha\mathsf{FedMgmt} \cup \alpha\mathsf{DeclMgmt} \cup \alpha\mathsf{ObjMgmt} \cup \alpha\mathsf{OwnMgmt}$$

$$NotPausedEvents = \alpha\mathsf{OwnMgmt} \cup \alpha\, ObjMgmt$$

$$FedEvents = \alpha\mathsf{SimInterface} \setminus \{ \begin{array}{l} \overline{\mathsf{joinFedExecution}}, \overline{\mathsf{resignFedExecution}}, \\ \overline{\mathsf{createFedExecution}}, \overline{\mathsf{destroyFedExecution}}\} \end{array}$$

$NotPausedEvents$ are those events which are not permitted during a pause. This list is a guess on our part, since *this is not specified in the IFSpec document*. The process ControlPause enforces this condition.

$FedEvents$ are those events which can only occur while the federate is a member of the federation execution and do not affect whether the federate is joined. This is less of a guess, since there are discussions in the IFSpec about what events are permitted while not joined (*i.e.*, one must obviously be permitted to join while not joined), but there are events, specifically those relating to queries, which I believe must be permitted while not joined in order to receive information. This contradicts the documentation, however. FedJoined enforces the preconditions regarding execution membership.

Both ControlPause and FedJoined control their events in the same way: They have marker events that delineate what state they are in, and when the events are permitted, they act as RUN_S, where S is the set of controlled events. The inclusion of RUN guarantees that the alphabet of these processes contains the events of interest, and so when they are placed in parallel with the other events, these events are constrained not to occur except when permitted.

Process SimInterface =
  FedMgmt $\|$ DeclMgmt $\|$ ObjMgmt $\|$ OwnMgmt
  $\|$ FedJoined $\|$ ControlPause
where
  $\mathsf{RUN}_S = \S \,\Box\, (\forall\, e : S \,\Box\, e \rightarrow \mathsf{RUN}_S)$
  $\mathsf{ControlPause} = \mathsf{RUN}_{NotPausedEvents}; \overline{\mathsf{pauseAchieved}} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \mathsf{ControlPause}$
  $\mathsf{FedJoined} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{RUN}_{FedEvents}; \overline{\mathsf{resignFedExecution}} \rightarrow \S$

B.2 RTI Glue Specification

SimInterface represents the constraints on the RTI interaction from the point of view of a single federate. The RTI connector specifies how the behavior of several federates is combined to create a complete federation execution. Thus, the RTI connector specifies one or more participating federates, each of which plays a Fed role in the connector.

The RTI has one auxiliary process that uses non-communication events: WhatSims. It uses an internal event, checkSims. This event is triggered internally to the RTI, and is never observed by any federate. The purpose of the WhatSims process is to keep track of the current membership of the federation, since this is a major factor in what events are permitted. For example, it is critical that the RTI never send an event to a member that is not joined. This creates the illusion of a "dynamic" system, since any federate that is not mentioned in the state variable of WhatSims might as well not exist. When a federate is created, it immediately sends a joinFedExecution event, after which it is included in the active state.

The other processes defined for the RTI handle actual interactions by the federates. The only protocol that affects the *control* state of the RTI is federation creation and destruction, represented by FedExists. The other protocols are basically independent, because any constraints on the RTI are either data-oriented or guaranteed by the behavior of the individual federates.

An example data-oriented constraint is the requirement that only federates that subscribe to an attribute are notified of its change. These constraints are represented in this model as non-determinism in the RTI. A data model of the RTI could make these constraints explicit, thus resolving the non-determinism.

An example of a constraint that is guaranteed by the individual federates is that there will be no object updates after pause has been achieved. This holds even though the RTI doesn't pay attention to pauses (beyond notifying federates) because the federates won't initiate events during a pause, and the RTI only initiates events as a response to federates.

Connector RTI($nsims$:1..)=
  Role $\textsf{Fed}_{1..nsims}$ = SimInterface
  Glue = FedExists
    $\parallel$ $\textsf{WhatSims}_{\{\}}$ $\parallel$ HandleSims
  where
  FedExists = $(\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.createFedExecution} \rightarrow \S)$;
    $((\forall\, i : 1..nsims \parallel \textsf{Run}_{\textsf{Fed}_{i:FedEvents}})$;
    $(\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.destroyFedExecution} \rightarrow \S))$
  $\textsf{WhatSims}_S$ = (checkSims!S $\rightarrow \textsf{WhatSims}_S$)
    $\Box\, (\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.joinFedExecution} \rightarrow \textsf{WhatSims}_{S\cup\{i\}})$
    $\Box\, (\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.resignFedExecution} \rightarrow \textsf{WhatSims}_{S-\{i\}})$
    $\Box\, \S$
  PauseProtocol = $\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.requestPause} \rightarrow$ checkSims?S
    $\rightarrow (\forall\, i : S;\ \overline{\textsf{Fed}_i\textsf{.schedulePause}} \rightarrow \S)$;PauseProtocol
  ResumeProtocol = $\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.requestResume} \rightarrow$ checkSims?S
    $\rightarrow (\forall\, i : S;\ \overline{\textsf{Fed}_i\textsf{.scheduleResume}} \rightarrow \S)$;ResumeProtocol
  SaveFed = $\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.scheduleFedSave} \rightarrow$ checkSims?S
    $\rightarrow (\forall\, i : S;\ \overline{\textsf{Fed}_i\textsf{.startFedSave}} \rightarrow \S)$;$\rightarrow$SaveFed
  RestoreFed = $\forall\, i : 1..nsims \,\Box\, \textsf{Fed}_i\textsf{.requestRestore} \rightarrow$ checkSims?S
    $\rightarrow (\forall\, i : S;\ \overline{\textsf{Fed}_i\textsf{.restore}} \rightarrow \S)$;$\rightarrow$RestoreFed

$\text{FedQuery} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{submitQuery} \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{query}} \rightarrow \S); \rightarrow \overline{\text{Fed}_i.\text{queryResult}} \rightarrow \text{FedQuery}$
$\text{HandleSubscribe} = \forall\, i : 1..nsims;\ e : \{subscribeObjClass, subscribeObjAttr\} \;[]\; \text{Fed}_i.e$
$\quad \rightarrow \text{checkSims?S}$
$\quad \rightarrow ((\forall\, i : S \sqcap \overline{\text{Fed}_i.\text{controlUpdates}} \rightarrow \S) \sqcap \S);\ \text{HandleSubscribe}$
$\text{HandleSubscribeInteraction} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{subscribeInteraction}$
$\quad \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{controlInteractions}} \rightarrow \S);$
$\quad \text{HandleSubscribeInteraction}$
$\text{NewObject} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{instantiateObj} \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{instantiateDiscoveredObj}} \rightarrow \S);$
$\quad \text{NewObject}$
$\text{DeleteObject} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{deleteObj} \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{removeObj}} \rightarrow \S);$
$\quad \text{DeleteObject}$
$\text{NewAttrValue} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{updateAttrValues} \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{removeObj}} \rightarrow \S);$
$\quad \forall\, K : \mathbb{P}\, S \setminus J \sqcap (\forall\, k : K; \overline{\text{Fed}_k.\text{instantiateDiscoveredObj}} \rightarrow \S);$
$\quad \forall\, L : \mathbb{P}\, S \setminus J \setminus K \sqcap (\forall\, l : L; \overline{\text{Fed}_l.\text{reflectAttrValues}} \rightarrow \S);$
$\quad \text{NewAttrValue}$
$\text{CancelObj} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{cancelObjReflection} \rightarrow \overline{\text{Fed}_i.\text{removeObj}} \rightarrow \text{CancelObj}$
$\text{Interaction} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{sendInteraction} \rightarrow \text{checkSims?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_j.\text{receiveInteraction}} \rightarrow \S);$
$\quad \text{Interaction}$
$\text{RequestAttr} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{requestAttrValueUpdate} \rightarrow \text{checkSims?S}$
$\quad \rightarrow ((\forall\, i : S \sqcap \overline{\text{Fed}_i.\text{provideAttrValueUpdate}} \rightarrow \S) \sqcap \S);\ \text{RequestAttr}$
$\text{OwnDivestiture} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{reqAttrOwnDivestiture} \rightarrow \text{checkSim?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_i.\text{reqAttrOwnAssumption}} \rightarrow \S);$
$\quad ((\overline{\text{Fed}_i.\text{attrOwnDivestitureNotifyRelease}}$
$\qquad \rightarrow \forall\, j : J \sqcap \overline{\text{Fed}_j.\text{attrOwnAcquisitionNotifySecured}} \rightarrow \S)$
$\quad \sqcap \overline{\text{Fed}_i.\text{attrOwnDivestitureNotifyRetain}} \rightarrow \S);$
$\quad \text{OwnDivestiture}$
$\text{UncOwnDivestiture} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{uncAttrOwnDivestiture} \rightarrow \text{checkSim?S}$
$\quad \rightarrow \forall\, J : \mathbb{P}\, S \sqcap (\forall\, j : J; \overline{\text{Fed}_i.\text{reqAttrOwnAssumption}} \rightarrow \S);$
$\quad ((\forall\, j : J \sqcap \overline{\text{Fed}_j.\text{attrOwnAcquisitionNotifySecured}} \rightarrow \S)$
$\quad \sqcap \S);$
$\quad \text{UncOwnDivestiture}$
$\text{OwnAcquisition} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{reqAttrOwnAcquisition} \rightarrow \text{checkSim?S}$
$\quad (\forall\, j : S \sqcap \overline{\text{Fed}_i.\text{reqAttrOwnRelease}} \rightarrow \S);$
$\quad (\overline{\text{Fed}_i.\text{attrOwnAcquisitionNotifySecured}} \rightarrow \S)$
$\quad \sqcap \overline{\text{Fed}_i.\text{attrOwnAcquisitionNotifyRejected}} \rightarrow \S);$
$\quad \text{OwnAcquisition}$
$\text{DeleteAcquisition} = \forall\, i : 1..nsims \;[]\; \text{Fed}_i.\text{reqDeletePrivAcquisition} \rightarrow \text{checkSim?S}$
$\quad (\forall\, j : S \sqcap \overline{\text{Fed}_i.\text{reqDeletePrivRelease}} \rightarrow \S);$
$\quad (\overline{\text{Fed}_i.\text{deletePrivAcquisitionNotifySecured}} \rightarrow \S)$

$\sqcap$ $\overline{\text{Fed}_i\text{.deletePrivAcquisitionNotifyRejected}} \rightarrow \S$);
DeleteAcquisition

OtherEvents = $\forall\, i : 1..nsims;\ e : \{pauseAchieved, resumeAchieved,$
$fedSaveBegun,$
$fedSaveComplete, restoreComplete$
$publishObjClass, publishObjAttr,$
$publishInteractionClass,$
$idRequest, queryAttrOwnership$
$\}$
$\square$ Fed$_i$.$e \rightarrow$OtherEvents

HandleSims = PauseProtocol $\parallel$ ResumeProtocol
$\parallel$ SaveFed $\parallel$ RestoreFed $\parallel$ FedQuery
$\parallel$ HandleSubscribe $\parallel$ HandleSubscribeInteraction
$\parallel$ NewObject $\parallel$ DeleteObject
$\parallel$ NewAttrValue $\parallel$ CancelObj
$\parallel$ Interaction $\parallel$ RequestAttr
$\parallel$ OwnDivestiture $\parallel$ UncOwnDivestiture
$\parallel$ OwnAcquisition$\parallel$ DeleteAcquisition
$\parallel$ OtherEvents

The process OtherEvents provides for all events that do not trigger other events. It simply recognizes them. (These events would change the state of the RTI in a model that used parameters.)

Bibliography

[AA96] Ahmed A. Abd-Allah. *Composing Heterogeneous Software Architectures*. PhD thesis, University of Southern California, August 1996.

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[AAG95] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–64, October 1995.

[AD94] Gregory D. Abowd and Alan J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. In *Proceedings of the SIGSOFT '94: 2nd ACM SIGSOFT Symposium on the Foundations of Sofware Engineering*, pages 44–52, New Orleans, LA, December 1994.

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of IFIP'92*, pages 134–41. Elsevier Science Publishers B.V., September 1992.

[ATT93] AT&T. *Best Current Practices: Software Architecture Validation*. AT&T, 1993.

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

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, New York, NY, 1996.

[Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1993.

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

[Bro96] Peter Brooks. New directions in advanced distributed simulation, April 1996. Presentation at CMU.

[BWW88] Mario R. Barbacci, C. B. Weinstock, and J. M. Wing. Programming at the processor-memory-switch level. In *Proceedings of the Tenth International Conference on Software Engineering*, March 1988.

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

[CA95] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, January 1995.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Systems Research Center, November 1989.

[Che88] C. H. Chen. *Signal Processing Handbook*. Marcel Dekker, Inc., New York, 1988.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. Technical Report CS-85-14, Brown University, 1985.

[DAR90] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden Valley, PA, July 1990. Software Engineering Institute.

[DG90] Norman Delisle and David Garlan. Applying formal specification to industrial problems: A specification of an oscilloscope. *IEEE Software*, 7(5):29–37, September 1990.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[DMSO] DMSO. Web site, URL = http://www.dmso.mil/docslib/hla/.

[DMSO95] DMSO. *Department of Defense High Level Architecture For Simulations Interface Specification*, October 1995. Version 0.2.

[DoD83] United States Department of Defense. *Reference Manual for the Ada Programming Language*, January 1983.

[FDR93] *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems (Europe) Ltd., Oxford, England, 1.3 edition, August 1993.

[G+95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm. On the definition of software system architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 85–94, Seattle, WA, April 1995.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT '94: 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–88, New Orleans, LA, December 1994. ACM Press.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

[Gar95] David Garlan, editor. *First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.

[Gar96] David Garlan. Style-based refinement for software architecture. In *Second International Software Architecture Workshop (ISAW-2)*, pages 72–75, San Fransisco, October 1996. ACM SIGSOFT.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An architecture description interchange language. submitted for publication, January 1997.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[GPT95] David Garlan, Frances Newberry Paulisch, and Walter F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, Feb 1995. Reprinted in ACM Software Eng. Notes, July 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–74, June 1987.

[Hav90] Klaus Havelund. *An RSL Tutorial*. Computer Resources International A/S, 1990.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Edinburgh University, March 1986.

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

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

[HS87] Roger Hayes and Richard D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13(12):1254–64, December 1987.

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

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, Reading, MA, 1992.

[Jac95a] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–89, October 1995.

[Jac95b] Michael J. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. ACM Press, New York, 1995.

[JC94] G.R. Ribeiro Justo and P.R. Freire Cunha. Deadlock-free configuration programming. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.

[JD96] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–95, July 1996.

[Jon86] C.B. Jones. Systematic program development. In *Proc. Symposium on Mathematics and Computer Science*, 1986. (also in *Software Specification Techniques*, pages 89-108).

[K$^+$] Gregor Kiczales et al. Aspect-oriented programming. Position Paper. http://www.parc.xerox.com/spl/projects/aop/position.htm.

[KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. SAAM: A method for analyzing the properties of software architecture. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1997.

[Kra90] Jeff Kramer. Configuration programming – a framework for the development of distributable systems. In *Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90)*, Israel, May 1990.

[KW96] Jeff Kramer and Alexander Wolf, editors. *Eight International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

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

[Lam87] David Alex Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.

[LeM96] Daniel Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–22, San Fransisco, CA, October 1996.

[LK97] Cristina Vieira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1997.

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

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

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

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

[Min91] Naftaly H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, 17(2):183–95, February 1991.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, San Fransisco, CA, October 1996.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95: Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, Software Engineering Notes 20(4), pages 18–28, Washington, DC, October 1995. ACM Press.

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

[MQ94] Mark Moriconi and Xiaolei Qian. Correctness and composition of software architectures. In *Proceedings of SIGSOFT '94: 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 164–74, New Orleans, LA, December 1994.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to software architectures. Submitted for Publication, January 1997.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of OOPSLA '93, ACM Sigplan Notices*, 28(10):1–15, October 1993.

[OMG91] Object Management Group. The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

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

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

[Pra86] V. R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

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

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premeriani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RD95] Jose Rivera and Alejandro Andres Danylyszyn. Formalizing the uni-processor simplex architecture. Technical Report CMU-CS-95-224, Carnegie Mellon University School of Computer Science, 1995.

[Rei90] S.P. Reiss. Connecting tools using message passing in the Field Environment. *IEEE Software*, 7(4):57–66, July 1990.

[Ros95] A. W. Roscoe. *Notes on CSP*. Unpublished manuscript, 1995.

[RR86] G. M. Reed and A. W. Roscoe. A time model for communicating sequential processes. In *Thirteenth International Colloquium on Automata, Languages, and Programming*, pages 249–262, 1986. Published in Theoretical Computer Science 58, 1988.

[SC] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. *Manuscript*.

[SC96] Mary Shaw and Paul Clements. Toward boxology: Preliminary classification of architectural styles. In *Second International Software Architecture Workshop (ISAW-2)*, pages 50–54, San Fransisco, October 1996. ACM SIGSOFT.

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

[SG94] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, 1994. Also printed as CMU Software Engineering Institute Technical Report SEI-94-TR-23, ESC-TR-94-023.

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

[Sha88] Mary Shaw. Toward higher-level abstractions for software systems. In *Proc. Tercer Simposio Internacional del Conocimiento y su Ingerieria*, pages 55–61, October 1988.

[SKW83] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–95, November 1983.

[SLF90] Reid Simmons, Long-Ji Lin, and Christopher Fedor. Autonomous task control for mobile robots. In *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, Philadelphia, PA, September 1990.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[SSM97] Kevin J. Sullivan, John Socha, and Mark Marchukov. Using formal methods to reason about architectural standards. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997. *To appear*.

[SVK93] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-base objects. Technical Report CMU-RI-TR-93-11, Carnegie Mellon University Robotics Institute, July 1993.

[Tic79] Walter F. Tichy. Software development control based on module interconnection. In *Proceedings of the Third International Conference on Software Engineering*, pages 29–41. IEEE Computer Society Press, May 1979.

[UML] Rational Software Corporation. Unified modeling language (UML). available at http://www.rational.com/uml/.

[WG97] Zhenyu Wang and David Garlan. A case study in architecture interchange. In preparation, 1997.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. MIT Press, Cambridge, MA, 1990.

[WKE93] J. Ward, J. Krueger, and E. Engstrom. ArchEd: A visual graph-based approach to architecture specification. Technical Report CS-R93-008, Honeywell Technology Center, March 1993.

[XOp93] X/Open Company Ltd. *Distributed Transaction Processing: Reference Model, Version 2*, November 1993.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[ZJ93] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.