# Formalism and Informalism in Software Architectural Style: a Case Study*

Robert J. Allen

Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213

*Proceedings of the First International Workshop on Architectures for Software Systems*, April 1995.

## 1 Introduction

Architectural style is increasingly recognized as an important element of software architectural design. An expanding body of research (*e.g.* [GS93, PW92, DAR90, MQ94, AAG93]) is recognizing that when developing a particular system, designers tend not to explore all possible alternatives for its architecture. Instead, they use specific structures and idioms that are effective for the domain in which they are working. We term such a collection of patterns and idioms an architectural *style*.

These patterns and idioms constrain the design space, permitting developers to ignore complications and alternatives that are not relevant to the system they are developing. By focusing on fewer, relevant alternatives, the issues that are most important are exposed and the developer is thus helped to make effective choices and locate a solution more easily. In addition, by guaranteeing properties of all systems constructed using the style, the cost of analyzing these systems is reduced.

For example, in "Deadlock-free Configuration Programming" [JC94], Justo and Cunha describe a collection of *component templates* and connectors that, when used to construct a system, can provide guarantees about the deadlocking behavior (or, preferably, the deadlock-freedom) of a configured system. By restricting the form of design elements permitted as components in their systems, they reduce the cost of the deadlock analysis to a simple static check.

Unfortunately, with few exceptions current exploitation of architectural style is informal and ad hoc [DAR90]. The choice of style in a system is often based on the development team's shared

*This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the U.S. Government, or Siemens Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

intuitions and experience rather than on any precise definition. For example, a style will often form around a collection of runtime support libraries that have been found to be useful. These libraries will be used based on informal guidelines and rules of thumb without there ever being any high-level characterization of the system structure that is supported by these usage patterns.

In this paper we consider an alternative to this ad hoc description of architectural styles, a formal description based on the inherent structure of software architectures. We explore this solution by describing Justo and Cunha's message passing style in WRIGHT, an architecture description language based on CSP [Hoa85]. We further show how the formalization of this style permits us to prove and extend the deadlock-freedom properties described in their paper. In the conclusion we evaluate the formal description and discuss some of the tradeoffs between formal and informal descriptions of software architectural styles.

## 2 An Overview of the Justo-Cunha Message Passing Style

A common way of configuring distributed systems is to use buffered message passing between distinct processes. An important problem that arises in these systems is the avoidance of deadlock. Because components are described, and often developed, independently, *a priori* analysis of deadlock is often difficult, or even impossible. As noted in the introduction, Justo and Cunha describe a specialization of this style [JC94] in which deadlock analysis is simplified to a static check for a cycle in the configuration.

Informally, the style consists of a single *connector type*, a pair of *component templates*, and a set of *configuration rules*. The connector type is a *message buffer*. The message connector is a finite buffer that provides non-blocking sends and blocking receives. If the buffer receives a message while it is full, the oldest message is discarded.

While all components in the style represent processes that send and receive via message buffers, the style further divides the legal components into two kinds, characterized by code templates. The first template, the *S-R* template, is characterized by a loop that first sends a message to each of its output ports, in order, and then receives a message on each of its input ports. The second template, the *R-S* template, reverses the send-then-receive pattern, first receiving a message on each input port and then sending a message on each output port.

In order to use these templates effectively, Justo and Cunha provide three configuration rules. These rules say, in effect, that in systems consisting of a cycle of template processes, if there are only R-S components then the system will deadlock; if not, then no deadlock will occur. Intuitively, we can see that this is true because deadlock can occur only when all components are waiting for a message. While R-S components may wait 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 these rules are clear in the simple case of a single cycle of all R-S components, it is not as clear how or whether these rules can be applied to more complex system configurations. In the next section we show how a formal description of the style permits the configuration rules to be made more flexible. We will further prove that the new rules correctly predict deadlock in any system constructed using the Justo-Cunha message passing style.

```
Connector Message(n:1..)
  role Sender = send → Sender
  role Receiver = receive → Receiver
  glue = $buf_0$
       where
         $buf_0$ = Sender.send → $buf_1$
         $buf_n$ = Sender.send → $buf_n$
              □ Receiver.receive → $buf_{n-1}$
         $buf_i$ = Sender.send → $buf_{i+1}$
              □ Receiver.receive → $buf_{i-1}$, $1 \leq i \leq n - 1$
```

Figure 1: The Message Buffer Connector in WRIGHT.

## 3 A Formal Description of the J-C Message Passing Style

We now consider a WRIGHT specification of the J-C message passing style. Through this example we will present the three elements of a WRIGHT style specification, component types, connector types, and configuration rules. We will show how formalizing a style resolves ambiguities in the informal specification and how analysis of the style, in particular through configuration rules, permits precise, low-cost analysis of the class of systems covered by the style.

The formal specification of the J-C message passing style includes the message buffer connector type and a characterization of the S-R and R-S templates. Given these specifications as a base, we discuss the configuration rules presented by Justo and Cunha, showing how we can generalize the templates and extend the deadlock analyses to a simple inductive rule.

### 3.1 Connector Types

The first 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.

While Justo and Cunha describe the interactions of their style informally, WRIGHT permits a precise specification of the interactions that will occur. 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 1. The general semantics of connectors is discussed in greater depth in [AG94a] and [AG94b].

The first part of a connector type specification, the *signature*, defines the name of the connector type and formal parameters to indicate the kinds of variability of the interaction. In general, the parameters could permit different numbers of participants and variability in the kinds of interactions that are represented by the connector. In the current example, the only dimension of variability that Justo and Cunha specify is in the size of the finite buffer. The buffer size is indicated by the parameter *n* in the connector specification.

The second part, the *role* specifications, indicates the participants in the interaction and the constraints on those participants. The process specified for each role indicates the communication actions that the participant may take, the ordering of those actions, and what participant in the interaction controls the sequencing of those events at any point in time. Because Justo and Cunha only consider non-terminating computations, the roles shown in figure 1 are simple repetitions of a single event. In general, however, they might show which participant in the interaction decides when to terminate and how this termination is achieved. For example, if we wanted to indicate that the `Sender` is not obligated to send an infinite number of times, we would add a *choice* of termination:

$$\textbf{role}\ \text{Sender} = \text{send} \rightarrow \text{Sender} \sqcap \surd$$

The receiver, on the other hand, would be obliged to *offer* the possibility of termination, but may also choose to terminate on its own:

$$\textbf{role}\ \text{Receiver} = (\text{receive} \rightarrow \text{Receiver} \,\square\, \surd) \sqcap \surd$$

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.

The main difference in our current description of connector types from earlier versions of WRIGHT is in the use of parameters to increase their flexibility. In earlier versions, it would have been necessary to fix the size of the buffer. Although it is not used in this example, it is also possible to vary the number of participants in an interaction. For example, we could permit more than one recipient for messages in a single buffer.

**Analysis of the Message Connector**

Having defined the connector type formally, we can begin to analyze the style both for consistency and for other properties. WRIGHT provides automated checks on connectors for internal deadlock freedom and on *conservatism* [AG94b]. Conservatism is a property of the glue that guarantees that the connector definition covers all situations permitted by legal configurations that use the connector. Because the roles always permit all events and the connector glue always permits at least one event, the message connector is both conservative and deadlock free for any legal value of the parameter *n*.

## 3.2 Component Types

The second part of a WRIGHT style definition is the introduction of component types. These provide a basis for the individual computations that will be composed into complete systems.

The structure of a component definition in WRIGHT is similar to that of a connector. Figure 2 shows the S-R template specification, while figure 3 shows the R-S template.

A component specification, like a connector, contains a signature that shows the variability of the component. 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*. Instead of role specifications, a component defines its *ports*. The ports indicate the different interactions in which the component will participate. Notice the symmetry between ports and roles, which indicate the set of components that will participate in a single interaction. In the J-C message passing style, an *in* port will fill a *receiver* role and an *out* port will fill a *sender* role.

**Component** S-R(r:1..;s:1..)
    **port** in(1..r) = receive $\rightarrow$ in
    **port** out(1..s) = send $\rightarrow$ out
    **comp spec** = S(1)
        **where**

$$S(i) = \begin{cases} \text{out}_i\text{.send} \rightarrow S(i+1), & 1 \leq i \leq s \\ \text{in}_{i-s}\text{.receive} \rightarrow S(i+1), & s < i < r + s \\ \text{in}_r\text{.send} \rightarrow S(1), & i = s + r \end{cases}$$

Figure 2: S-R Template

**Component** R-S(r:1..;s:1..)
    **port** in(1..r) = receive $\rightarrow$ in
    **port** out(1..s) = send $\rightarrow$ out
    **comp spec** = R(1)
        **where**

$$R(i) = \begin{cases} \text{in}_i\text{.receive} \rightarrow R(i+1), & 1 \leq i \leq r \\ \text{out}_{i-r}\text{.send} \rightarrow R(i+1), & r < i < r + s \\ \text{out}_s\text{.send} \rightarrow R(1), & i = s + r \end{cases}$$

Figure 3: R-S Template

In a component specification, the *comp spec* indicates how the component's various interactions are combined to form a distinct computation. It is in the `comp spec` 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.

**Analysis of the Component Types**

As with the connector type, we can ask if the component types are internally consistent. A component is internally consistent if it satisfies its port specifications and is deadlock-free. The component specifications will eventually produce an infinite number of all receive and send events, and so both templates meet these requirements.

## 3.3 Configuration Rules

The final part of a style description, the part that ties it together, is the collection of configuration rules. These rules provide global constraints on systems and expose properties and analyses that are important to the successful design of systems using the defined style.

In this section we will show how formalizing the informal deadlock avoidance rules presented by Justo and Cunha clarifies the analysis in two important ways. First, the scope of the rules must be made more precise. In what situations do the rules apply, and where do they need to

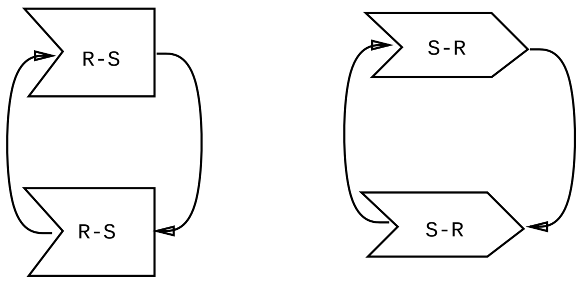

Figure 4: Counter example to deadlock claim, rule 1

be generalized before they can be used? Second, by formalizing the rules their validity can be determined more rigorously. Are the rules as stated correct? If not, can we restate them in a form that permits us to prove their correctness?

In their description of the message passing style, Justo and Cunha provide three configuration rules [JC94, page 152]:

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.

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. Consider the following potential rephrasing of rule 1:

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

This seemingly reasonable interpretation of the rule turns out to be incorrect, as the system in figure 4 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 rule is now provable, by observing the traces of the components, and noting that each R-S component requires an initial send event to be produced by some other component before it can itself produce a send event. Because no component can be first, none 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 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 component will block forever waiting for a send from the R-S cycle (which we know from rule 1 will never happen).

Again we can ask if there is a reasonable restriction of rules 2 and 3 that is both correct and clarifies over what systems they apply. One such restriction is the following:

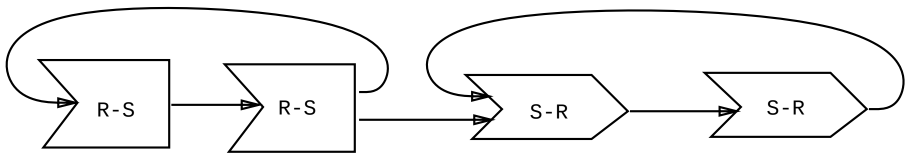

Figure 5: Counterexample to rule 2

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

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 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.

## 4 Conclusion

In this paper we have shown how a style that has been presented informally can be formalized using the architectural specification language WRIGHT. Having formalized the style as it was presented informally, we were able to demonstrate a useful analysis of the style and to prove more carefully a key property of the style, deadlock-freedom.

Having carried out this exercise, the question arises: What are the advantages and disadvantages of a formal characterization relative to an informal one? I would argue that in fact both kinds of characterization are useful and that a complete exploration of a style must include both kinds. 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 |
| • Based on architectural intuitions | • Based on architectural principles |

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.

## References

[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.

[AG94a] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, volume 29(8). SIGPLAN Notices, August 1994.

[AG94b] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[DAR90] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden Vallen, PA, July 1990. Software Engineering Institute.

[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. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

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

[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.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 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.