Formal Modeling and Analysis of Architectural Standards

Robert J. Allen
IBM, Dept. AQPV / 862F
1000 River Street
Essex Junction, VT 05452 USA
(802) 769-5934
roballen@vnet.ibm.com

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
(412) 268-5056
garlan@cs.cmu.edu

James Ivers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
(412) 268-1590
jivers@cs.cmu.edu

ABSTRACT
An important trend in commercial software development is the creation of architectural standards that describe a common reference architecture for a family of related applications. Currently architectural standards are typically described using informal, or semi-formal, techniques, such as application programming interface specifications, implementation guidelines, and box-and-line diagrams. In this paper we show how formal architectural modeling and analysis can be applied to architectural standards. In particular, we use the recently-issued High Level Architecture (HLA) Standard for Distributed Simulation to illustrate how architectural specification can expose several important classes of architectural design flaws, including errors of omission, design inconsistencies, potential deadlocking behavior, and race conditions.

KEYWORDS
Software architecture, architectural standards, reference architectures, formal Specification, HLA, Wright

1 INTRODUCTION

Architectural frameworks are increasingly being recognized as a significant point of leverage in the development of software systems. Architectural frameworks typically determine the structure of a family of applications, providing shared infrastructure and prescribing requirements for instantiating the framework to produce a particular application. Often architectural frameworks are developed as in-house proprietary systems that permit the rapid development of new applications in a product line [13, 15]. However, they are also used to define open integration standards that permit multiple vendors to contribute parts to produce a composite system, or to provide components that can interact smoothly with those supplied by other vendors. Examples of such frameworks include the ISO OSI layered protocol stack and the CORBA object integration framework.

Architectural frameworks introduce new challenges not faced by traditional one-of-a-kind systems. First, they are partial. Architectural frameworks are by their nature incomplete. Hence, they cannot be tested (or often even implemented) in isolation. Second, their design impacts numerous (as yet undeveloped) systems. Thus a design error in the framework can propagate to the potentially very large number of systems constructed with it. This raises the stakes for assuring that the design is sound and will meet its intended benefits. Third, frameworks must cope with adaptation. While traditional systems may evolve slowly over time, architectural frameworks are intended to be frequently instantiated and adapted to meet the needs of specific applications. This means that a framework must permit variability along certain explicit dimensions.

These challenges lead to a number of specific problems: How do you characterize architectural frameworks—making explicit the allowed dimensions of variability and assumed commonality? How do you state properties that the framework will guarantee to hold for any valid instantiation? How do you state the requirements on the parts that are composed with the framework? How can you test for implementation conformance–both of the shared framework infrastructure and interfaces to externally-produced components that are integrated with the framework? How can you debug the framework design in the absence of specific instantiations?

In response to questions like these, there has been a lot of recent research activity in the area of architectural description and analysis of software systems. In particular, the research community has developed a number of software architecture description languages (ADLs),[1] architectural development tools and environments, and techniques for architectural analysis.

Most of these languages and tools, however, focus on the problem of representing the architectures of *individual*

Draft: Submitted for Publication, September 1997.

[1] There are currently over a dozen such languages.

systems. In particular, although ADLs differ considerably in the kinds of analyses they support, they typically assume that all system components are known, and that the primary challenge is to determine the properties of that specific collection of components.

In this paper we consider the problem of modeling and analyzing *architectural frameworks* – and especially architectural standards. As we will show, architectural specification can be instrumental in defining an architectural standard precisely, and in detecting flaws in the design. To make this concrete we will illustrate the techniques by describing how the Wright ADL was applied to the High Level Architecture (HLA) for Distributed Simulation, an architectural standard recently proposed by the US Defense Modeling and Simulation Office. Specifically, we will show how formal architectural analysis can be used to determine sources of ambiguity, incompleteness, and internal inconsistency.

After outlining related work, we will begin by giving an overview of the HLA and the challenges it raises. Then we briefly describe Wright to show how it represents system architectures. The goal here is not to provide a detailed explanation of Wright, but rather to give the flavor of the language and its approach to architectural specification and analysis. Next we explain how we mapped parts of the RTI specification into Wright. We then describe a selection of the insights that we gained using tools for analyzing Wright. Finally we discuss the general principles to be gleaned from the approach.

2 RELATED WORK

The primary claim of this paper is that formal modeling of architectures can be highly instrumental in clarifying architectural standards, and in discovering flaws in their design. As such, it is related to two distinct areas of related research.

The first area is the growing field of architectural description and analysis. Currently there are many ADLs and tools to support their use (such as [9, 12, 7, 11, 10]). While ADLs are far from being in widespread use, there have been several examples of their application to realistic case studies.

This paper contributes to this body of case studies, but pushes on a different dimension – namely, the application of architectural modeling to standards. As noted above, standards introduce new challenges for architectural description that are not addressed by looking at single systems.

The second area is research on the analysis of frameworks and standards. An example close in spirit to our work is that of Sullivan and colleagues, who used Z to model and analyze the Microsoft COM standard [14]. Also closely related, is work on formal definitions of architectural styles. In particular, Moriconi and colleagues describe techniques for refining between styles [11]. In other work carried out by this paper's authors, we have considered how Z can be used to define styles [1]. Similarly, we have looked at several small case studies of style using Wright [4, 4]. This work differs in that it represents a much larger scale application of architectural modeling than has been reported in the literature, and that it deals explicitly with an architectural standardization effort.

3 THE EXAMPLE AND ITS CONTEXT

The High Level Architecture (HLA) is an example of an industrial-strength architectural standard. It attempts to provide an integration standard for distributed simulations, essentially defining a kind of "simulation bus" into which simulations developed by different vendors can be plugged and then interoperate. Specifically, the HLA standard prescribes the interface requirements that must be met by simulation writers, and provides a Run Time Infrastructure (RTI) design to handle the coordination and communication for an ensemble of such simulations.[2] (See figure 1.) As a standardization effort the HLA has undergone extensive review by numerous agencies and advisory committiees, prototyping efforts, and public review (the standard is accessible on the Web at http://www.dmso.mil/projects/hla/tech/ifspec). Although it will likely continue to undergo minor revisions over the coming years, vendors are currently actively building simulations that conform to the standard.

The HLA defines the coordination of individual simulations that are intended to communicate object attributes and events. In the HLA design, members of a *federation* – the HLA term for a distributed simulation – coordinate their models of parts of the world by 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 broadcasts updates using the facilities of the RTI. Communication 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, or *IFSpec*. Each such communication is provided by a service which is defined by a name, the initiator (Federate or RTI), a set of parameters, a possible return value, pre and post conditions, and a set of exceptions that may occur during execution of the service.

The interface is divided into six parts: federation management, declaration management, object management, ownership management, time management, and data

[2] We were working from version 1.2 of the interface specification, or *IFSpec*.

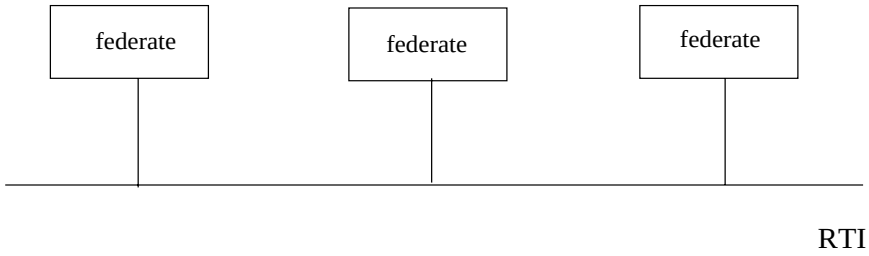

Figure 1: The HLA Architectural Standard

distribution management. Federation management services are used by federates to initiate a federation execution, to join or leave an execution in progress, to pause and resume, and to save execution state. Declaration services are used to communicate about what kinds of object attributes are available and of interest, while object services communicate actual object values. Ownership services 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.[3] The fifth category, time management, is 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. Data distribution management is used to filter attribute updates, reducing message traffic and processing requirements, for each federate based on defined criteria.

The intention 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. As part of the current standard development effort, several implementation efforts, each termed a *proto-federation*, are underway.

There are three obvious requirements that the interface specification should satisfy:

- **Interface sufficiency:** Are the service preconditions (that constrain federate behavior) sufficiently strong to guarantee that a correctly implemented RTI can preserve certain critical system invariants? Invariants include such things as that there should be at most one owner for every object attribute.
- **Relative Completeness:** As we noted above, frameworks like the HLA are by their nature incomplete. However, one can ask whether a framework is complete *relative* to the task it is performing. That is, is the framework specified completely enough that it is possible to build a correct implementation?
- **Understandability:** As a standard, it is critical that simulation creators understand what is expected of them and what kinds of behavior they can count on from the RTI. Does the documentation convey this adequately? Are there unnecessary ambiguities that make it difficult to determine what the real behavior of an RTI will be?

Unfortunately, the current form of the interface specification makes it difficult to determine whether these requirements are satisfied.

First, since each operation is specified in isolation, it is almost impossible to tell whether the preconditions will guarantee that valid *sequences* of operations can occur at run-time. For example, it is very difficult to answer questions like: what kind of behavior might possibly precede the invocation of service P? Similarly, it is almost impossible to tell whether multiple federates will have consistent views of the overall state of a federation (clearly a desirable property). For example, if one federate believes that the federation is paused, will all other federates (perhaps eventually) believe this, too?

Second, the specification does not explicitly indicate what aspects of the design are *intentionally left out*. This makes it difficult to evaluate whether missing information is an oversight on the part of the specifiers or something that must be explicitly provided by an actual implementation of the RTI. Moreover, in the case of the latter, the requirements of that elaboboration are not clear.

Third, the specification is largely informal. This makes it hard to pin down specific effects of service invoca-

[3] Example situations include when the original federate must drop out, 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 federates responsible for modelling troops in each region might hand off ownership of the unit's representation object.

tionss. Also, it does not indicate what the dynamic behavior of the system will be. In particular, the allowable or expected *sequences* of calls are never described, but must be inferred implicitly by reasoning about when the preconditions of operations will be satisifed by some preceding sequence of service calls.

We will illustrate in the remainder of the paper how a formal specification of the HLA in Wright can improve the situation.

4 WRIGHT

Wright is a formal language for describing software architecture. As with most architecture description languages, Wright describes the architecture of a system as a collection of interacting components. However, unlike many languages, Wright supports the explicit specification of new architectural connector types and architectural styles.

To illustrate, a simple Client-Server system description is shown in Figure 2. This example shows three basic elements of a Wright system description: style declaration, instance declarations, and attachments. The instance declarations and attachments together define a particular system configuration.

An *architectural style* is a family of systems with a common vocabulary and rules for configuration. A simple style definition is illustrated in Figure 3. This style defines the vocabulary for the system example of Figure 2. As we will see later, a style can also define topological constraints on systems that use the style.

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 defines the set of possible interactions in which the component may participate.

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. 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. The glue defines how the roles will interact with each other.

Each part of a Wright description – port, role, computation, and glue – is defined using a variant of CSP [8].[4] For example, a simple client role might be defined by the CSP process:

$$\textbf{Role}\ \text{Client} = (\overline{\text{request}} \rightarrow \text{result?x} \rightarrow \text{Client}) \sqcap \S$$

[4] In this paper we will only be able to briefly describe the notation. Details of the semantic model and the supporting toolset can be found elsewhere [3, 5, 2].

```
Configuration SimpleExample
Style ClientServer
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

Figure 2: A Simple Client-Server System

```
Style ClientServer
    Component Server
        Port Provide [provide protocol]
        Computation [Server specification]
    Component Client
        Port Request [request protocol]
        Computation [Client specification]
    Connector C-S-connector
        Role Client [client protocol]
        Role Server [server protocol]
        Glue [glue protocol]
end ClientServer.
```

Figure 3: A Simple Client-Server Style

This defines a participant in an interaction that repeatedly makes a request and receives a result, or chooses to terminate successfully.

As is partially evident from this example Wright extends CSP in some minor syntactic ways. First, it distinguishes between *initiating* an event and *observing* an event. An event that is initiated by a process is written with an overbar: The specification of the Client's Request port would use the event $\overline{\text{request}}$ to indicate that it initiates a request. The Server's Provide port, on the other hand, waits for some other component to initiate a request (it *observes* the event), so in its specification this event would be written without an overbar: request.

Second, a special event in Wright is $\surd$, which indicates the successful termination of a computation. Because this event is not a communication event, it is not considered either to be initiated or observed. Typically, use of $\surd$ occurs only in the process that halts immediately after indicating termination: $\S = \surd \rightarrow \text{STOP}$.

Third, to permit parameterization of connector and component types, Wright uses a quantification operator: $\langle op \rangle\ x : S \bullet P(x)$. 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, $\Box\ i : \{1,2,3\} \bullet P_i = P_1 \Box P_2 \Box P_3$. A special case is $;\ x : S \bullet P(x)$, which is some unspecified sequencing of the processes: $;\ x : S \bullet P(x) = \sqcap\ x : S \bullet (P(x)\ ;\ (;\ y :$

$S \setminus \{x\} \bullet P(y)))$.

As discussed in [5], descriptions of connectors can be used to determine whether the glue constrains the roles enough to guarantee critical properties such as local absence of deadlock. These descriptions can also be used to determine whether a configuration is properly constructed, *e.g.,* whether the interfaces of a component are appropriate for use in a particular role. But these issues are beyond the scope of this paper.

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

5 FORMALIZING THE HLA RTI

Turning now to the HLA RTI, the Wright formalization has focussed on specifying the IFSpec as a style. That makes sense because the HLA is a guideline for clarifying the construction and behavior of many different federations. Each federation would be a configuration in the "HLA Style". (Or rather, the parts of a federation that are selected for a particular federation execution would be such a configuration.)

The basic elements of the HLA formalization consist of the introduction of a single component type, the *federate*, and a single connector, the *RTI*. In addition, there is a configuration constraint rule specifying that there shall be a single RTI connector and all federates shall interact using it.

While the overall specification is considerably larger than can be shown in a short paper, a few extracts will give the flavor. The overall Wright specification of the HLA style (without details) is shown below.

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

To specify the properties that are required of any federate to participate in an HLA simulation, an *interface type* is introduced, *FederateInterface*, that defines what the communication behavior of the federate will be. The FederateInterface introduces the various service invocations that will pass between the federate and the RTI. Services (represented by *events* in Wright) are divided into those that are initiated by a federate, such as $\overline{\mathsf{joinFedExecution}}$, which indicates that the federation wishes to participate in the simulation, and those that are initiated by the RTI, such as reflectAttributeValues, which is used to inform a federate of new data values. Recall that the presence of an overbar (as in $\overline{\mathsf{e}}$) indicates an event that is *initiated* by the process. An undecorated event (as in e) indicates an observation of the activity of some other process. An extract of the *FederateInterface* definition is as follows:

Interface Type FederateInterface = JoinFed
  $\sqcap\ \overline{\mathsf{createFedExecution}} \rightarrow$ JoinFed
  where
    JoinFed = $\overline{\mathsf{joinFedExecution}} \rightarrow$ NormalExecution
    NormalExecution = IntiateFedActivity $\square$
      WaitForFedActivity $\sqcap$ EndFedMgmt
    EndFedMgmt = $\overline{\mathsf{resignFedExecution}} \rightarrow$
      (§ $\sqcap\ \overline{\mathsf{destroyFedExecution}} \rightarrow$ §)

This extract indicates that before joining an execution, the federate may need to create it (if no other federation has), and that it must indicate the start of computation by an explicit joinFedExecution service. The federate is then in the condition NormalExecution, where it can both invoke services on the RTI and permit the RTI to invoke services on it. Finally, the federation may, during normal execution, choose to resign from the execution (indicated by the resignFedExecution service within the EndFedMgmt process), after which it must not invoke any services on the RTI or permit any services on it to be invoked.

While the FederateInterface models the behavior of a single federate, the RTI describes how multiple federates interact. In the connector specification, the **Glue** provides a specification indicating how events of one component relate to those of the others. In the extract in figure 4, event names are prefixed with Fed_i to indicate that it is an event of the *ith* federate.

This extract of the RTI connector specification clarifies the specification in FederateInterface that each federate has the option of creating the RTI execution: Exactly one of them must do so, and none of the others are permitted to do so. Similarly, the RTI execution must not be destroyed unless there are no joined federates, and once the RTI is destroyed, no further interaction may take place.

6 ANALYSIS

Specification of the HLA has intrinsic benefits insofar as it provides a precise statement of the standard, focus-

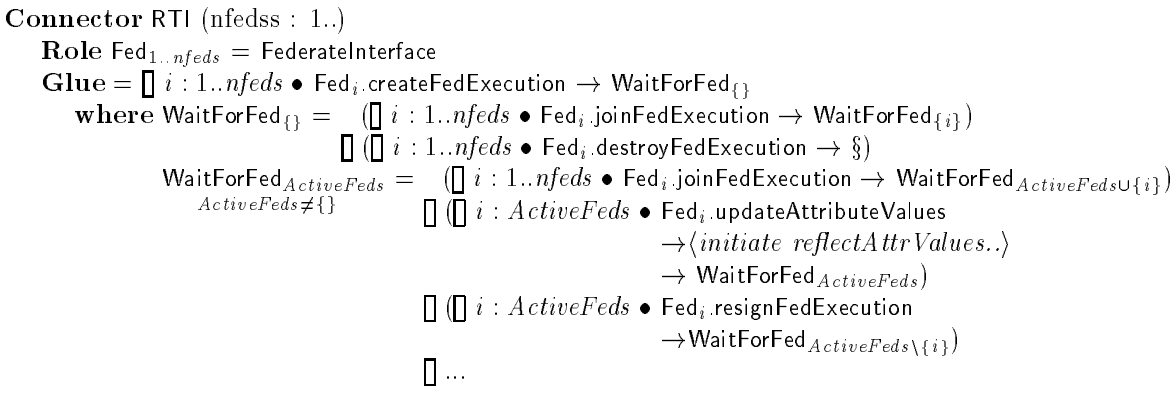

Figure 4: An extract of the RTI Connector.

ing on dynamic behavior of the RTI and its connected federates.

We can also use the specification as a basis for formal analysis. In particular, it is possible to apply formal tools to gain additional insight. We will illustrate this idea with three examples that were detected using the Wright toolset on the specification.

Creation of the execution: Our first example discovery concerns the start up behavior of a federation execution. As illustrated earlier, when it starts, a federate must decide whether to create the execution, and then, before invoking any other services, it must join the federation.

The corresponding part of the RTI is as follows:

ConnectorRTI()
Glue = $\Box\ i : 1..nfeds \bullet \mathsf{Fed}_i$.createFedExecution →
$\mathsf{WaitForFed}_{\{\}}$
where $\mathsf{WaitForFed}_S$ =
$\Box\ i : 1..nfeds \bullet \mathsf{Fed}_i$.joinFedExecution →
$\mathsf{WaitForFed}_{S \cup \{i\}}$

The specification states that the first event must be a createFedExecution from any one of the federates. After this service has been invoked, the RTI is in the state WaitForFed, in which it is possible for all of the federates to invoke joinFedExecution. Note how after createFedExecution the process' control state changes (to WaitForFed) 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 (none of which may occur before the create).

Trouble arises with the trace represented in figure 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. It represents an omission in the IFSpec, because there is a precondition, that a federate must not create the execution if it already exists, but no way for a federate to discover the information it needs to satisfy the precondition.

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 federation execution does not exist" as a precondition. The service joinFedExecution has a corresponding precondition "The 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.

Paused on join: In the previous example, Wright analysis exposed 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 infor-

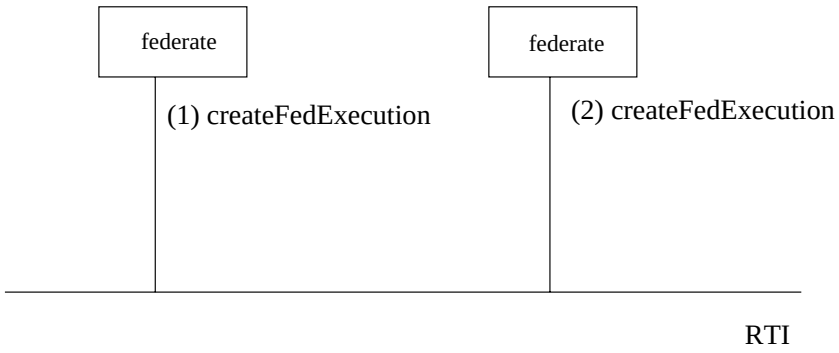

Figure 5: Oops! Deadlock when two federates create

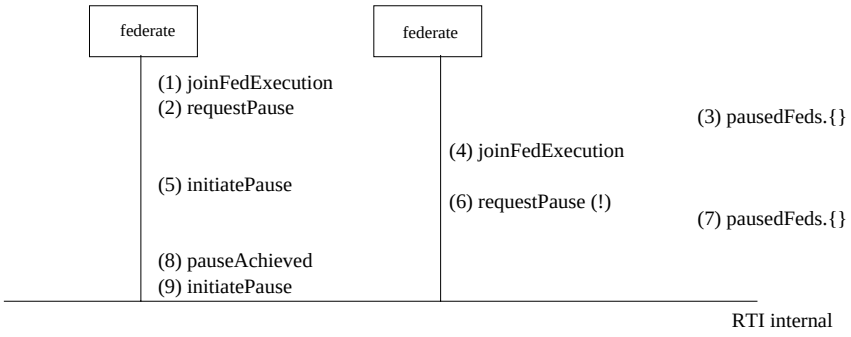

Figure 6: Another Deadlock: federates are confused about pausing

mal documentation.[5]

For the example above, deadlock occurs immediately, or after at most two events. This isn't a very deep insight; any development effort could not get very far without stumbling across this situation. However, this kind of insight should not be dismissed as trivial since it represents only 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 FederateInterface specification:

$$
\begin{aligned}
&\mathsf{JoinFed} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{NormalExecution} \\
&\mathsf{NormalExecution} = \mathsf{IntiateFedActivity} \,\Box\, \mathsf{WaitForFedActivity} \\
&\qquad\qquad \sqcap\ \mathsf{EndFedMgmt} \\
&\mathsf{InitiateFedActivity} = \overline{\mathsf{requestPause}} \rightarrow \mathsf{NormalExecution} \\
&\mathsf{WaitForFedActivity} = \mathsf{initiatePause} \rightarrow \\
&\qquad\qquad \overline{\mathsf{pauseAchieved}} \rightarrow \mathsf{FedPaused} \\
&\mathsf{FedPaused} = (\overline{\mathsf{requestResume}} \rightarrow \mathsf{FedPaused}) \\
&\qquad\qquad \sqcap\ \mathsf{PauseWait} \\
&\mathsf{PauseWait} = \mathsf{initiateResume} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \\
&\qquad\qquad \mathsf{NormalExecution}
\end{aligned}
$$

This extract focuses on the pause and resume behavior of a federate. It indicates that in the NormalExecution 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 initiated. Once a pause is initiated, the federate pauses itself, notifies the RTI of its success, and is then in the state FedPaused. This is the inverse of NormalExecution — in this state, it does not carry out normal events, but instead may request a resume (but not another pause), and should expect that an initiateResume will occur. Once it does, the federation is in the running state again.

The RTI glue shows how these events are combined in different federates through mini-protocols such as PauseProtocol:

$$
\begin{aligned}
&\mathsf{PauseProtocol} = \\
&\quad \mathsf{PauseRequests}_S \parallel \mathsf{ResumeRequests}_S \parallel \mathsf{PausedFeds}_{\{\}} \\
&\mathsf{PauseRequests}_S = \\
&\quad \Box\, i : S \bullet \mathsf{Fed}_i.\mathsf{requestPause} \rightarrow \mathsf{pausedFeds?T} \rightarrow \\
&\quad (;\, i : (S \setminus T) \bullet \overline{\mathsf{Fed}_i.\mathsf{initiatePause}} \rightarrow \S)\ ; \\
&\quad \mathsf{PauseRequests}_S \\
&\mathsf{PausedFeds}_S = \\
&\quad (\Box\, i : 1..nfeds \bullet \overline{\mathsf{pausedFeds!S}} \rightarrow \mathsf{PausedFeds}_S \\
&\quad \Box\ (\Box\, i : 1..nfeds \bullet \mathsf{Fed}_i.\mathsf{pauseAchieved} \rightarrow \\
&\qquad\quad \mathsf{PausedFeds}_{S \cup \{i\}}) \\
&\quad \Box\ (\Box\, i : 1..nfeds \bullet \mathsf{Fed}_i.\mathsf{resumeAchieved} \rightarrow \\
&\qquad\quad \mathsf{PausedFeds}_{S \setminus \{i\}})
\end{aligned}
$$

[5] Our tools are based in part on a commercial model-checker for CSP, called FDR.

This indicates that whenever a federate requests a pause, all joined federates which are not already paused will receive a notification via the initiatePause service. PausedFeds is used to keep track of which federates are currently paused. The corresponding resume mini-protocol, represented by the ResumeRequests process, is not shown yet.

This protocol of pause and resume results in a problem as depicted in figure 6. Deadlock arises because after event 6, Fed_2.requestPause, there is a race condition regarding what will happen next. As shown, event 7, pausedFeds, occurs before Fed_1 can report that a pause has been achieved. This causes the RTI to then want to again direct the first, now paused, federate to intiatePause in event 9, but according to Fed_1 the next event will be Fed_1.initiateResume. If, however, events 7 and 8 had been reversed, this problem would not arise. The RTI would correctly recognize that the federate has achieved a pause and would not attempt to issue a second initiatePause. The problem with this sequence is that when it joins the federation execution, Fed_2 *doesn't know the system is paused* and can initiate unanticipated activity.

It is worth emphasizing that this scenario is complicated enough 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 services 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. Wright found this property even though we weren't looking for it in particular and we didn't know it was there.

Deadlocked execution: Our third example points out the possibility that a federation execution designed to the HLA standard can become deadlocked. This example again deals with the pause and resume protocols; specifically, a single federate can prevent the entire federation from resuming execution by simply refusing to pause.

The previous extract from the FederateInterface specification is changed as follows to model the ability of a federate to refuse to pause:

$$\begin{aligned}\mathsf{WaitForFedActivity} = \mathsf{initiatePause} \rightarrow (\\ (\overline{\mathsf{pauseAchieved}} \rightarrow \mathsf{FedPaused}) \sqcap\\ (\overline{\mathsf{cannotPause}} \rightarrow \mathsf{NormalExecution}))\end{aligned}$$

Adding the specification for the resume mini-protocol to the previously illustrated glue specification illustrates the problem with this:

$$\begin{aligned}&\mathsf{ResumeRequests}_S =\\ &\quad \Box\, i : S \bullet \mathsf{Fed}_i.\mathsf{requestResume} \rightarrow \mathsf{pausedFeds?T} \rightarrow\\ &\quad \mathsf{RequestResumeResponse}_{S==T,S}\\ &\mathsf{RequestResumeResponse}_{true,S} =\\ &\quad (;\, i : S \bullet \overline{\mathsf{Fed}_i.\mathsf{initiateResume}} \rightarrow \S)\ ;\\ &\quad \mathsf{ResumeRequests}_S\\ &\mathsf{RequestResumeResponse}_{false,S} = \mathsf{ResumeRequests}_S\end{aligned}$$

The boolean condition to RequestResumeResponse leads directly back to a precondition of the requestResume service which states "The federation execution is paused." In order for the federation execution to be paused, each federate which is a member of the federation must be paused. Therefore, the ability of a federate to refuse to pause thus leads directly to the possibility that a federation execution can become deadlocked.

7 IMPACTS

We can identify several impacts of formalization.

First, a key property of the approach is that by formalizing the HLA as an architectural style, the associated analysis of our specification informs us about properties of the IFSpec in general, not of any particular proto-federation. That is, if we discover a property of the specification and prove that it holds, it must hold for every federation that obeys the IFSpec. If one of the proto-federation efforts discovers a problem in their implementation, it is difficult to tell whether it is fundamental to the IFSpec, permitted by the IFSpec but not necessarily true of every implementation, or an indication that the prototype implementation is in violation of the IFSpec. With the Wright specification, the analysis will indicate whether the property is intrinsic or only a possibility. Because the specification is formal, it is possible to verify that the specification does indeed obey the IFSpec (*e.g.,* that the preconditions of a servicee are satisfied whenever that service is invoked).

The main impact of our formalization effort is on the IFSpec itself. By providing an analysis of the properties of the IFSpec, we can help determine whether the IFSpec ensures the properties that are desired and discover inconsistencies or other weaknesses of the specification.

The impact of the formalization on the IFSpec can occur in two places. First, it can help suggest places where the RTI standard needs to be changed or strengthened; and second, it can provide a basis for supplemental documentation or indicate where the documentation might be elaborated even when the standard itself does not need to be changed.

As an example of the latter, consider "exceptions." Part of each service definition in the IFSpec is a list of ex-

ceptions. For example, joinFedExecution includes the exception "federate already joined." In our attempt to formalize the HLA, we realized that the formalization (and presumably any implementation) wasn't possible unless we knew if these exceptions resulted in actual message traffic or whether they were simply anomalies that should be considered (but without explicit notification).

Examples of the former come in two ways. First, the act of formalizing the IFSpec and providing precise semantics for each service uncovers numerous ways in which the standard can be strengthend. In particular, using Wright (along with an auxiliary Z specification), has identified several categories of opportunities for improvement, such as:

- ambiguities
- missing information
- insufficient pre and postconditions

Second, once we have determined enough detail about the specification to formalize it, Wright can be used to detect potential problems, such as:

- unanticipated consequences of services
- race conditions (like the Paused on Join example)

Examples of ambiguities and missing information are numerous (more than thirty have been identified). In many cases, this is because the standard is sufficient to deal with the typical run-time scenario, but lacks the precision to describe how unusual occurences must be handled. For example, when a federate saves its state, it associates the save with a save label. State can be restored through a restore service, but state can only be restored when all federates have a save for the save label being restored. However, there is no mention of how long this save label can be successfully used; after what point can a federate discard a previous save state?

An interesting example of an insufficient precondition concerns the above mentioned restore service. The standard includes a service which the RTI uses to tell a federate to restore its state. The precondition to this operation stated "The federation has a save with the specified label." The problem with this is that even if the federation does have such a save, a given federate may not (e.g., it might have joined the execution after the save took place). This precondition has since been changed to the correct form which states "The federate has a save with the specified label."

As an example of unanticpated consequences which were brought to light by the Wright analysis, there is a case in which as soon as a federate creates an object it is informed by the RTI that it should remove that object. The sequence of services and conditions which cause this obviously undesirable consquence is a subtle one which was not exposed by looking at individual services. It was the examination of the sequence of small consequences of each individual service which illustrated this case.

One impact of this kind of analysis is to find a category of problems that should not be solved in the standard, but should be discussed in some kind of supplementary documentation. For example, if a federation execution achieves a paused state, no progress can be made unless some federate requests that execution be resumed. However, there is no requirement in the standard that federates do this, as there shouldn't be. This is a case where a problem has been illustrated by our analysis whose solution is outside the realm of the standard. In this particular case, it is the responsibility of those using the HLA to resolve this problem. Supplementary documentation should point out such trouble spots and, where possible, point out possible solutions from which an HLA client should select. In this case, users of the HLA merely need to establish some consistent policy of ending pauses (such as, whichever federate requested the pause is responsible for requesting the resume, or designating some federate as the one that always must request resumes) in order to avoid an execution which gets stuck because no one requests a resume.

8 DISCUSSION AND CONCLUSION

This paper has described an approach to formalization and analysis of architectural standards, using the HLA as an example. To carry this out, we employed four techniques that have general utility for architectural modeling and analysis.

First, was the translation of the published architectural standard expressed from a set API function calls into a description in which legal sequences of calls is made explicit. This required examining the pre-conditions for each call, and determining how that call could be sequenced with other calls in the API. By putting it in this form (here expressed in the CSP subset of Wright), we were able to check for anomolous situations, not easily detectable from the original specification.

Second, was the use of abstraction to make the architectural specification tractable (both intellectually and for our model-checking tools). In particular, to do this we abstracted away the details of the data model. While this led to a lack of precision it greatly simplified the overall specification. (The current on-line specification runs about 150 pages, while the Wright specification is 15 pages long.)

Third, was making careful distinction between parts of the standard that characterize the required behavior of

the participants in an interaction (namely, the individual simulations), and the behavior that combines those behaviors into system wide interactions (namely, the RTI itself). In Wright this distinction is captured by the difference between connector roles and glue, but has analogues in other ADLs that support first class connectors. The ability to distinguish these two concerns helps manage the complexity of the standard and to isolate the problems when they were detected.

Fourth, was a careful attention to structuring the architectural specification to match the published standard's structure. In particular, we divided our specification into parts that directly corresponded to the "management groups" in the IFSpec. By doing this we were able to partition our effort into incremental steps (tackling one management group at a time), and to have a high degree of traceability back to the original document.

It is important to note, however, that such formalizations are just one of many tools and notations that are needed. Wright is good at detecting certain kinds of anomalies – primarily those associated with protocols of interaction. But there are many other issues that are not addressed, such as real-time behavior, state models, and compliance testing. This suggests that future work on modeling architectural standards can and should exploit other complementary approaches to architectural modelling and analysis.

ACKNOWLEDGEMENTS

A short, early version of this paper was presented in [6], and much of the initial research was carried out as part of one of the author's Ph.D. work [2].

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation Grant CCR-9357792. 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 US Department of Defense, the United States Government, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans SE Eng and Methodology*, Oct. 1995.

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, SCS, Jan. 1997. CMU/SCS Report CMU-CS-97-144.

[3] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[4] R. Allen and D. Garlan. A case study in architectural modelling: The AEGIS system. In *Proc of the 8th Intl Workshop on SW Specification and Design (IWSSD-8)*, Mar. 1996.

[5] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[6] R. Allen and D. Garlan. Formal modeling and analysis of the HLA RTI. In *Summary Report of 1997 Spring Simulation Interoperability Wkshop*. Inst. for Simulation and Training, Mar. 1997.

[7] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc of SIGSOFT'94: The 2nd ACM SIGSOFT Symp on the Foundations of SW Engineering*. ACM Press, December 1994.

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

[9] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE TSE*, April 1995.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of the 5th European SW Eng Conference, ESEC'95*, Sep. 1995.

[11] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[12] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE TSE*, 21(4):314–335, April 1995.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] K. Sullivan, J. Socha, and M. Marchukov. Using formal methods to reason about architectural standards. In *Proc. of the International Conf. on SW Engineering (ICSE97)*, May 1997.

[15] W. Tracz. DSSA frequently asked questions. *Software Engineering Notes*, 19(2):52–56, April 1994.