FORMAL MODELING AND ANALYSIS OF THE HLA RTI*

Robert Allen
David Garlan

Carnegie Mellon University
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

KEYWORDS
Software Architecture, Formal Specification, HLA, Wright

ABSTRACT

The HLA RTI is a complex artifact, supporting several classes of interaction (e.g., federation management, object management, time management). A critical challenge in producing an RTI architectural framework (and its associated simulation interface specifications) is to develop confidence that its specification is well-formed and complete. In this paper we describe on-going work in formally modelling the HLA both to document the standard more precisely, as well as to analyze it for anomalies, omissions, inconsistencies, and ambiguities. The technical basis for this work is the use of a formal architectural description language, called Wright, and its accompanying toolset.

1 INTRODUCTION

Architectural frameworks are rapidly 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. 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.

The High Level Architecture (HLA) is in this latter category. It attempts to provide an integration standard for distributed simulations, prescribing the interface requirements that must be met by simulation writers, and providing a Run Time Infrastructure (RTI) design to handle the coordination and communication for an ensemble of such simulations.

Architectural frameworks, such as the HLA, 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

*In *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, Orlando FL, March 3-7, 1997.

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,[1] architectural development tools and environments, and techniques for architectural analysis. Although it is too early to tell whether they will be successful, proponents argue that these notations, tools, and techniques can have a major impact on architectural development for real systems.

In this paper we describe our initial experinece in applying one such architectural description language, called Wright, to the HLA RTI. Working from an early draft of the RTI Specification (Version 0.2 of the IFSpec) we represented several parts of the RTI in formal architectural terms. We then performed several analyses on the specification to determine sources of ambiguity, incompleteness, and internal inconsistency.

We will begin by giving an overview of 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 strengths and limitations of the approach.

2 OVERVIEW AND RATIONALE

The HLA defines a standard for the coordination of individual simulations through the communication of data object attributes and events [3]. 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 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, or *IFSpec*. 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 management messages 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 messages are used to communicate about what kinds of object attributes are available and of interest, while 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.[2] 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.

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 message 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

[1] At last count there were over a dozen such languages.

[2] 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.

that there should be exactly one owner of 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 procedure 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 calls. 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 RTI in Wright can improve the situation. For illustrative purposes our examples are drawn from an early version of the specification: we are just now starting an effort to apply the techniques to the most recent release.

```
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 1: A Simple Client-Server System

3 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 1. 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 2. This style defines the vocabulary for the system example of Figure 1. 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

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 2: A Simple Client-Server Style

CSP [4].[3] For example, a simple client role might be defined by the CSP process:

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

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 $\overline{\mathsf{request}}$ 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: $\S = \surd \rightarrow STOP$.

Third, to permit parameterization of connector and component types, Wright uses a quantification operator: $\forall x : S \langle op \rangle P(x)$. This operator constructs a new process based on a process expression and the set S, combining its parts by the operator $\langle op \rangle$. For example, $\forall i : \{1,2,3\} \mathbin{[\,]} P_i = P_1 \mathbin{[\,]} P_2 \mathbin{[\,]} P_3$. A special case is $\forall x : S\ ;\ P(x)$, which is some unspecified sequencing of the processes: $\forall x : S\ ;\ P(x) = \forall x : S \sqcap (P(x)\ ;\ \forall y : S \setminus \{x\}\ ;\ P(y))$.

As discussed in [2], 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.

4 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 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.

To specify the properties that are required of any federate to participate in an HLA simulation, an *interface type* is introduced, *SimInterface*, that defines what the communication behavior of the federate will be. The SimInterface introduces the various messages

[3] 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 [2, 1].

that will pass between the federate and the RTI. Messages (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 *SimInterface* definition is as follows:

$$
\begin{array}{l}
\textbf{Interface Type}\ \mathsf{SimInterface} = \mathsf{JoinFed} \\
\qquad\qquad\qquad\qquad \sqcap\ \overline{\mathsf{createFedExecution}} \rightarrow \mathsf{JoinFed} \\
\quad \textbf{where} \\
\qquad \mathsf{JoinFed} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{NormalExecution} \\
\qquad \mathsf{NormalExecution} = \mathsf{reflectAttributeValues} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathsf{NormalExecution} \\
\qquad\qquad\qquad\qquad\quad \square \ldots \\
\qquad\qquad\qquad\qquad\quad \sqcap\ \overline{\mathsf{resignFedExecution}} \rightarrow \S
\end{array}
$$

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 message. The federate is then in the condition NormalExecution, where it can both send and receive messages from the RTI. Finally, the federation may, during normal execution, choose to resign from the execution (indicated by the resignFedExecution message), after which it must not send or receive any more messages.

While the SimInterface 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 3, 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 SimInterface 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.

5 ANALYSIS

Specification of the HLA RTI has intrinsic benefits insofar as it provides a precise statement of the design, focusing 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 two examples that were detected using the Wright toolset on the early release of the specification.[4]

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 sending any other messages, it must join the federation.

The corresponding part of the RTI is as follows:

$$
\begin{array}{l}
\textbf{Connector}\mathsf{RTI}() \\
\quad \textbf{Glue} = \forall\, i : 1..nsims\ \square\ \mathsf{Fed}_i.\mathsf{createFedExecution} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{WaitForSim}_{\{\}} \\
\quad \textbf{where}\ \mathsf{WaitForSim}_S = \\
\qquad \forall\, i : 1..nsims\ \square\ \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{WaitForSim}_{S \cup \{i\}}
\end{array}
$$

The specification states that the first event must be a createFedExecution from any one of the federates. After this message is received, the RTI is in the state WaitForSim, in which it is possible for all 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 4. 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,

[4] These specific problems have been corrected in recent releases of specification.

$$
\begin{array}{l}
\textbf{Connector } \mathsf{RTI}\ (\mathrm{nsims} : 1..) \\
\quad \textbf{Role } \mathsf{Fed}_{1..nsims} = \mathsf{SimInterface} \\
\quad \textbf{Glue} = \forall\, i : 1..nsims \ \square\ \mathsf{Fed}_i.\mathsf{createFedExecution} \rightarrow \mathsf{WaitForSim}_{\{\}} \\
\qquad \textbf{where } \mathsf{WaitForSim}_{\{\}} = \forall\, i : 1..nsims \ \square\ \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \mathsf{WaitForSim}_{\{i\}} \\
\qquad\qquad\qquad\qquad\qquad\quad \square\ \forall\, i : 1..nsims \ \square\ \mathsf{Fed}_i.\mathsf{destroyFedExecution} \rightarrow \S \\
\qquad\quad \underset{ActiveFeds \neq \{\}}{\mathsf{WaitForSim}_{ActiveFeds}} = \forall\, i : 1..nsims \ \square\ \mathsf{Fed}_i.\mathsf{joinFedExecution} \rightarrow \mathsf{WaitForSim}_{ActiveFeds \cup \{i\}} \\
\qquad\qquad\qquad\qquad\qquad\qquad \square\ \forall\, i : ActiveFeds\ \square\ \mathsf{Fed}_i.\mathsf{updateAttributeValues} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \langle send\ reflectAttrValues..\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{WaitForSim}_{ActiveFeds} \\
\qquad\qquad\qquad\qquad\qquad\qquad \square\ \forall\, i : ActiveFeds\ \square\ \mathsf{Fed}_i.\mathsf{resignFedExecution} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{WaitForSim}_{ActiveFeds \setminus \{i\}} \\
\qquad\qquad\qquad\qquad\qquad\qquad \square\ \ldots
\end{array}
$$

Figure 3: An extract of the RTI Connector.

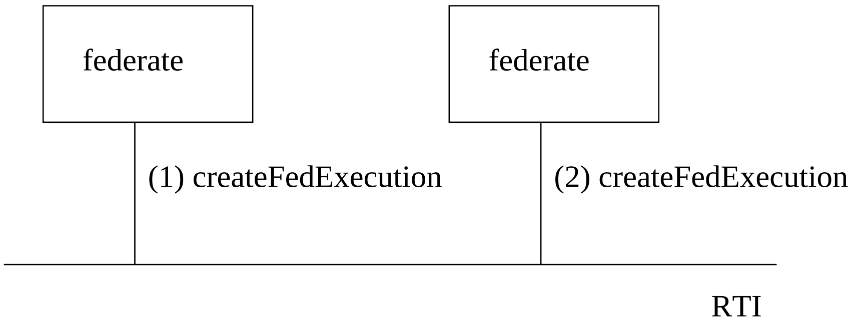

Figure 4: Oops! Deadlock when two federates create

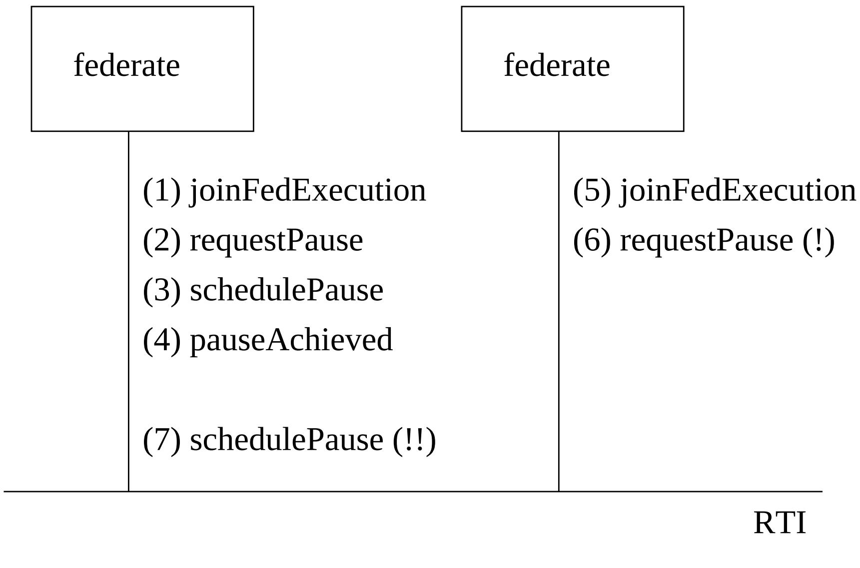

Figure 5: Another Deadlock: federates are confused about pausing

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.

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 informal 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 SimInterface specification:

$$\begin{aligned}
&\mathsf{JoinFed} = \overline{\mathsf{joinFedExecution}} \rightarrow \mathsf{ContFed}\\
&\mathsf{ContFed} = \overline{\mathsf{requestPause}} \rightarrow \mathsf{ContFed} \sqcap \mathsf{WaitForEvent}\\
&\mathsf{WaitForEvent} =\\
&\qquad\qquad \mathsf{schedulePause} \rightarrow \overline{\mathsf{pauseAchieved}} \rightarrow \mathsf{FedPaused}\\
&\mathsf{FedPaused} =\\
&\qquad\qquad \overline{\mathsf{requestResume}} \rightarrow \mathsf{FedPaused} \sqcap \mathsf{PauseWait}\\
&\mathsf{PauseWait} =\\
&\qquad\qquad \mathsf{scheduleResume} \rightarrow \overline{\mathsf{resumeAchieved}} \rightarrow \mathsf{ContFed}
\end{aligned}$$

This extract focuses 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 in the running state again.

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

$$\begin{aligned}
&\mathsf{HandlePause}_S =\\
&\quad \forall\, i : S \Box\ \mathsf{Fed}_i.\mathsf{requestPause} \rightarrow\\
&\qquad\quad (\forall\, i : S\ ;\ \overline{\mathsf{Fed}_i.\mathsf{schedulePause}} \rightarrow \S)\ ;\ \mathsf{HandlePause}_S
\end{aligned}$$

This indicates that whenever a federate requests a pause, all joined 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 5. 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.*

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 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. Wright found this property even though we weren't looking for it in particular and we didn't know it was there.

Once we 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 POTENTIAL IMPACTS

We can identify several potential impacts of formalization of the HLA.

First, a key property of the approach is that by formalizing the HLA as an architectural style, the asso-

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

ciated 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, there is no way 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 message are satisfied whenever that message is sent).

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 message definition in the IFSpec is a list of exceptions. 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).

Once we have determined enough detail about the specification to formalize it, Wright can be used to detect potential conflicts from, for example:

- insufficient preconditions
- missing information
- race conditions
- wrong directionality (parameter v. return value, RTI v. federate initiation)

As an example of missing information, consider the example of the destroyFedExecution message, which indicates that the current execution is done and that the RTI should terminate. This message has a precondition that ensures that this message is safe to execute: there must be no joined federates. But there is no way, using the IFSpec as written, for a federate to determine what the set of joined federates is. Thus, there must be some external way for a federate to get this information before sending the message. (For example, a federation might define a special "query" message, the federate might be a person who also controls starting and stopping all of the simulation federates, or the federate might simply ignore the precondition and count on getting an exception whenever there is anything still executing.)

One impact of this kind of analysis is to suggest a core set of query messages that all federations must supply. If that solution isn't satisfactory to everyone (which wouldn't be surprising), then some kind of supplementary documentation should discuss these kinds of issues ("the following ambiguities in the IFSpec are deliberate and must be resolved by any federation... Possible solutions include...") Obviously the formalization can't resolve the pragmatics of various options, but it does provide a means of exposing possible starting points.

7 CONCLUSION

This paper has briefly described an approach to formalization and analysis of the HLA RTI using Wright. While our early work is preliminary, it suggests that there is considerable merit in formalizing certain aspects of the architectural framework. Based on our early experience, we are now applying Wright to the most recent release of the specification, and attempting to broaden its coverage to more aspects of the RTI.

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 this topic can and should exploit other complementary approaches to architectural modelling and analysis.

ACKNOWLEDGEMENTS

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] Robert Allen. *Formalizing Software Architecture.* PhD thesis, Carnegie Mellon School of Computer Science, 1997. To appear.

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

[3] DMSO. Web site, URL = http://www.dmso.mil/docslib/hla/.

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