Formal Modeling and Analysis of the HLA Component Integration Standard

Robert J. Allen
IBM, Dept. AQPV / 862F
1000 River Road
Essex Junction, VT 05452 USA
roballen@btv.ibm.com

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
garlan@cs.cmu.edu

James Ivers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
jivers@cs.cmu.edu

ABSTRACT

An increasingly important trend in the engineering of complex systems is the design of component integration standards. Such standards define rules of interaction and shared communication infrastructure that permit composition of systems out of independently-developed parts. A problem with these standards is that it is often difficult to understand exactly what they require and provide, and to analyze them in order to understand their deeper properties. In this paper we use our experience in modeling the High Level Architecture (HLA) for Distributed Simulation to show how one can capture the structured protocol inherent in an integration standard as a formal architectural model that can be analyzed to detect anomalies, race conditions, and deadlocks.

KEYWORDS

Component integration standards, component-based software, protocol families, software architecture, formal specification.

1 Introduction

Component integration standards are becoming increasingly important for commercial software systems. The purpose of a component integration standard is to define rules of interaction and shared infrastructure for composing independently-developed software components into larger systems. Typically an integration standard prescribes requirements that must be satisfied by component interfaces, and it provides facilities that support communication and coordination among those components.

An early example of a component integration standard is Unix pipes, which requires components to have interfaces that read and write byte streams, and provides buffering and synchronization infrastructure to connect the components together. More recent examples include a growing number of domain-specific integration standards in areas as diverse as programming environments, robotics control [20], and signal processing [17]. Additionally some aspects of general-purpose object-oriented systems, such as CORBA, COM/DCOM/OLE/ActiveX, and JavaBeans function as component integration standards.

Component integration standards greatly simplify the construction of complex systems from existing parts. Since components share assumptions about the nature of interaction with their environment many of the general problems of component mismatch do not arise [9]. Thus it is easier for implementors to combine parts written by multiple vendors and to add new parts to existing systems. Moreover, the use of a standard's supporting infrastructure can substantially reduce the amount of custom code that must be written to support communication between those parts.

In practice, integration standards are typically specified using a combination of informal and semi-formal documentation. On the informal side are guidelines and high-level descriptions of usage patterns, tips, and examples. On the semi-formal side one usually finds a description of an application programmers' interface (API) that explains what kinds of services are provided by the infrastructure. APIs are formal to the extent that they provide precise descriptions of those services—usually as a set of signatures, possibly annotated with informal pre- and post-conditions.

While such documentation is necessary, by itself it leaves many important questions unanswered—for component developers, system integrators, standard infrastructure implementors, and proposers of new standards. For example, while it may be clear what are the names and parameters of services provided by the integration infrastructure, it may not be clear what are the restrictions (if any) on the ordering of invocations of those services. It may not be clear what kinds of run-time state is maintained by the infrastructure to facilitate component interaction. It may not be clear what facilities *must* be provided by a component to be a component, and which are optional. It may not be clear how concurrently executing components might impact each other's run-time behavior, particularly when they access shared resources. It may not be clear whether the standard itself contains latent design problems that can lead to unexpected runtime anomalies, such as race conditions and deadlocks.

In this paper we show how one can use formal modeling to clarify these kinds of issues. The key idea is to treat the integration standard as a structured protocol that can be analyzed using existing formalisms and tools for modeling software architecture. By making explicit the protocol inherent in the integration standard, we are able to make precise the requirements on both the components and on the supporting infrastructure itself. This in turn provides a deeper understanding of the standard, and supports analysis of its properties.

While the use of protocols to model a component integration standard might seem like a natural idea, there are a number of technical hurdles that make it non-trivial to do in practice. First, many component integration standards are relatively complex, often involving dozens of routines in their API. Structuring becomes a central issue for modeling. Second, for a complex standard it is critical that the formal model be traceable back to the original documentation. This is because when errors are found, it must be possible to relate the results back to the source. Third, is the issue of variability in the standard. It is critical to distinguish between aspects of the model that are fixed by the standard and those that are allowed to vary from one system to another. In practice this can be difficult to do because a particular API may make implementation choices that are not intrinsically part of the integration standard. Fourth is the problem of tractability. If the formal model is to be useful to humans or to analysis tools it must be simple enough that it can be understood (or mechanically processed), but detailed enough that useful properties are revealed.

In the remainder of this paper we describe our experience of solving these technical problems for a complex integration standard for distributed simulation. The primary contributions of this paper are twofold. First, we show how formal architectural models based on protocols can clarify the intent of an integration standard, as well as expose critical properties of it. Second, we describe the techniques that can be used to create the initial model, and structure it to support traceability, tractability, and automated analysis.

2 Related Research

This work is closely related to three distinct areas of prior research. The first area is the growing field of architectural description and analysis. Currently there are many architecture description languages (ADLs) and tools to support their use (such as [12, 19, 15, 14]). 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 component integration standards.

Among existing ADLs the one used here, Wright, is most closely related to Rapide [12], as both use event patterns to describe abstract behavior of architectures. Indeed, parts of the HLA have been modeled by the developers of Rapide. Wright differs from Rapide insofar as it supports definition of connectors as explicit semantic entities and permits static analysis using model checking tools. As we will see, this capability is at the heart of our approach for modeling integration standards.

The second related area is research on the analysis of 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 [21]. Also closely related is work on formal definitions of architectural styles. In particular, Moriconi and colleagues describe techniques for refining between styles [15]. In other work carried out by this paper's authors, we have considered how Z and Wright can be used to define styles [1, 2]. The work described in this paper differs from previous work in this area in that it represents a much larger-scale application of architectural modeling than has been reported in the literature, and introduces new techniques to carry it out.

The third area is work on protocol specification and analysis. There has been considerable research on ways to specify protocols using a variety of formalisms, such as I/O Automata [13], SMV [6], SDL [11], and Petri Nets [16]. While our research shares many of the same goals, there are notable differences. First, most protocol analysis assumes you are starting with a complete description of the protocol. In contrast, in our work the protocol is typically *implicit* in the API of some integration standard documentation. Second, while most protocols may involve large numbers of states, the number of entry points into the protocol is typically small. In contrast, the HLA (and other similar standards) has over a 125 different entry points. This leads to technical issues not typically dealt with in the protocol literature, such as ways to structure such a broad interface.

HLA	High Level Architecture
IFSpec	HLA Interface Specification
Federate	an individual simulation
Federation	a set of coordinated simulations
RTI	Run-Time Infrastructure
Service	A routine in the IFSpec

Figure 1: Glossary of HLA Terms

3 The "High Level Architecture" for Distributed Simulation

The "High Level Architecture" (HLA) is a component integration standard for distributed simulation [22]. It was developed by the Defense Modeling and Simulation Office (DMSO) to support interoperability between simulations purchased from different vendors.[1] This is a critical concern for the US government, which spends billions on third party simulations, coming from a wide variety of vendors.

Informally, the HLA prescribes a kind of "simulation bus" into which simulations can be "plugged" to produce a joint (distributed) simulation (as illustrated in Figure 2). 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 a runtime infrastructure (termed the *RTI*). Routines that support 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*. Routines, or "services", in the IFSpec are specified by a name, the initiator (either a Federate or the RTI), a set of parameters, a possible return value, pre- and post-conditions, and a list of the exceptions that may occur as a result of invoking the service. (Figure 1 summarizes the HLA terms used in this paper.)

An example of a typical RTI service is shown in Figure 3 (taken from [22]). This service is initiated by a federate (an individual simulation) when it wants to pause the federation (the entire distributed simulation). The effects of calling the service are to cause the RTI to coordinate a distributed handshaking algorithm in which it asks each of the simulations to pause.

The HLA is a complex integration standard. The current IFSpec includes over a 125 different services, and the full document is over 400 pages of description. While the part of the HLA design that deals with attribute broadcast is relatively straightforward, the overall standard is complicated significantly by the need to deal with issues such as starting, stopping, and pausing; allowing one federate to transfer object attribute ownership to another; and distributed clock management and time-ordered message sequencing.

To make the documentation for the integration standard manageable, the IFSpec is divided into six chapters: federa-

[1] This paper refers to Version 1.2 of the standard, issued August 1997. A more recent version (1.3 of April 1998), fixing numerous problems, was recently released. In addition the HLA is currently in the process of being revised as an IEEE standard (provisional number P1516) by the Simulation Interoperability Standards Organization (SISO).

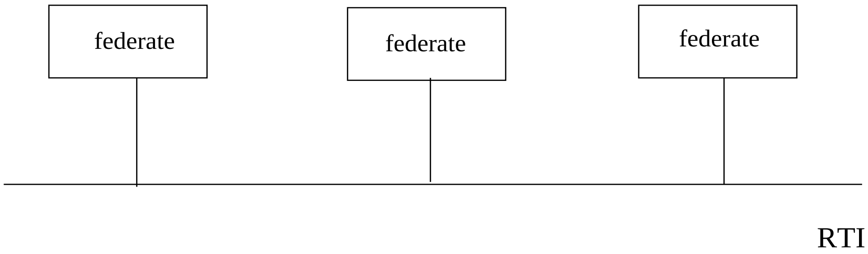

Figure 2: The HLA Integration Standard

2.5 Request Pause

Initiator: Federate-Initiated

Indicates to the RTI the request to stop the advance of the federation execution. The federation execution members will be instructed by the RTI to pause as soon after the inovcation of the Request Pause service as possible. The label, supplied when the pause is requested, will be supplied to the other federates via the Initiate Pause service.

Supplied Parameters
  A label
Returned Parameters
  None
Pre-conditions
  The federation execution exists
  The federate is joined to that federation execution
  The federation execution is advancing (not paused)
Post-conditions
  A federation pause is pending
Exceptions
  Federation already paused
  Federate not a federation execution member
  RTI internal error
  ...
Related Services
  Initiate Pause
  Pause Achieved
  ...

Figure 3: The RequestPause Service

tion management, declaration management, object management, ownership management, time management, and data 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. Time Management services are used to coordinate the logical time advancements of federates and to ensure that messages are delivered in time-stamp order. Data distribution management is used to filter attribute updates, reducing message traffic and processing requirements, for each federate based on defined criteria.

4 Problems with the IFSpec

The IFSpec is an indispensible document, since it provides a definition of each required and provided service of the standard. However, there a number of problems with using IFSpec as the *only* form of HLA documentation.

First, is the problem of determining what are the permissible or required orderings of service invocations. While the preconditions indicate (informally) in what situations a given service can be called, it is often hard to determine what kinds of behavior would lead to a precondition being satisfied or not. For example, is it always legal for a federate to pause the federation after joining? Moreover, it is difficult to tell whether the preconditions are complete: if a component satisfies all preconditions, will it ever trigger exceptions?

Second, since the IFSpec describes the HLA from the point of view of an individual simulation, it is difficult for someone building a federation out of existing simulations to tell what kinds of coordination behavior will be provided by the RTI. For example, exactly what kind of protocol does the RTI use to pause a federation? Does an RTI attempt to find an owner for orphaned object attributes?

Third, is the problem of understanding the deeper properties of the standard, both with respect to its intended behavior, and with respect to anomalies that can arise in using it. For example, are there sequences of service invocations that might lead to system deadlock? Are there latent race conditions or other sequences of events that can lead to anomalous behavior?

5 Approach and Challenges

In the remainder of the paper we show how a formal architectural specification can help resolve these kinds of issues. The keystone of the approach is to view the HLA as an architectural standard centered around a connector (i.e., the RTI) that permits simulation components (i.e., the federates) to interact with each other. We then provide a formal specification of that connector's behavior, thereby making explicit the protocol inherent in its informal description.

While the use of protocols might appear to be a natural idea, there are a number of technical challenges in specifying an integration standard as complex as the HLA.

- **Structure:** It is essential to structure the specification so that it permits (a) separation of concerns; (b) incremental specification and analysis; and (c) traceability. Separation of concerns is needed to manage complexity. Incremental specification is required to allow increasing levels of fidelity, depending on the needs for documentation and analysis. Traceability is needed so that issues identified in the process of formalization can be related to the source of the problem in the standard's API.
- **Abstraction:** Abstraction is required for two reasons. First, it is necessary to simplify the model so that it becomes tractable both for human readers and for analysis tools. Second, it is necessary to indicate what parts of the standard are (intentionally) left unspecified.
- **Analysis:** Once you have a formal definition, it is not immediately clear what kinds of analyses one would want to perform, or how to frame those analyses using existing tools.

Connector C-S-connector(n: Int)
Role $\text{Client}_{1..n}$ = ($\overline{\text{request}} \rightarrow$ result?x $\rightarrow$ Client) $\sqcap$ §
Role Server = (request $\rightarrow \overline{\text{result!x}} \rightarrow$ Server) $\square$ §
Glue =
($\square$ i: 1..n •
Client_i.request $\rightarrow \overline{\text{Server.request}} \rightarrow$
Server.result?x $\rightarrow \overline{\text{Client}_i\text{.result!x}} \rightarrow$ Glue)
$\square$ §

Figure 4: Simple Client-Server Connector

In the next two sections we present the model and discuss how it addresses these issues. Specifically, we use the Wright architectural description language (ADL) as the modeling language [4] to define the HLA. The key feature of Wright that we exploit is the ability to formally define new architectural connectors as structured protocols.

6 Wright

Like most ADLs, Wright defines a system as a composition of components and connectors: the components define the primary centers of computation, while the connectors define the interactions between components. Unlike some ADLs, however, Wright permits the explicit definition of new connector types, and provides formal, automatable criteria for checking the consistency of those types [4].[2]

In Wright a connector type has a *name*, an optional set of *parameters*, a set of *role* descriptions, and a *glue* description. The name identifies the kind of connector. The parameters provide instantiation values for the connector. Each role has a specification that defines the possible behaviors of a participant in the interaction. The glue defines how the roles will interact with each other.

To illustrate, consider a client-server connector that permits multiple clients to interact with a server. Figure 4 shows how this might be written in Wright. The connector has a parameter that determines the number of clients that can access the server. The roles of the connector define how the clients and servers must behave at their interfaces. The glue specifies how client-server communication is coordinated.

The distinction between roles and glue in Wright is important because it allows us to separate two quite different concerns of the connector specification. First is the description of the *interfaces* to the connector: each role identifies what an individual participant must do to interact over that kind of connector. Second is the specification of how the connector *coordinates* those participants. As we will see in the case of the HLA, this separation allows us to distinguish between the interface that each simulation must conform to, and the coordinating behavior of the run-time infrastructure.

Wright uses a variant of CSP [10] to define role and glue behavior. Each such specification defines a pattern of events (called a process) using operators for sequencing ("$\rightarrow$" and ";"), choice ("$\sqcap$" and "$\square$"), and parallel composition ("$\|$"). Appendix A contains more details on the parts of CSP that we use in this paper.

Wright extends CSP in three 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. Second, it uses the symbol § to denote the successfully-terminating process. (In CSP this is usually written "SKIP".) Third, Wright uses a quantification operator: $\langle op \rangle\ x : S \bullet P(x)$. This operator constructs a new process based on the process expression $P(s)$, and the set S, combining its parts by the operator $\langle op \rangle$. For example, $\square\ i : \{1,2,3\} \bullet P_i = P_1 \square P_2 \square P_3$: i.e., a choice among one of three processes, P_1, P_2, or P_3. Similarly, $;\ x : S \bullet P(x)$, is a process that consists of some unspecified sequencing of the processes: $;\ x : S \bullet P(x) = \sqcap\ x : S \bullet (P(x)\ ;\ (;\ y : S \setminus \{x\} \bullet P(y)))$.

Refering again to Figure 4, the process defining the Server role of the C-S-connector

$$\text{Server} = (\text{request} \rightarrow \overline{\text{result!x}} \rightarrow \text{Server}) \square \S$$

indicates that the server repeatedly either observes a request and then initiates the output of a result (represented by variable x), or else terminates. Since we use the CSP operator for "external" choice ($\square$), the decision about whether to terminate or accept a request is determined by the environment of the server.[3]

The connector also defines n client roles. Each of the n roles has the same behavior:

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

indicating that the client can repeatedly initiate a request and retrieve a result, or it can choose to terminate. In this case we use the internal choice operator ($\sqcap$) to indicate that, unlike the server, it is the client's choice whether to terminate the interaction.

Finally, the glue part of C-S-connector coordinates the clients and servers by forwarding requests and returning results. (In this case the glue has a particularly simple behavior: we'll see later that this need not be the case.) The glue guarantees that a complete request-reply transaction between a given client and the server will complete before accepting another request. The use of the quantification (using $\square$) requires the glue to wait for some client to make a request. If several do so simultaneously, the glue is free to pick one. Note that in the description of the glue, we tag each event with the name of the role with which it is associated. Also note that initiated events from roles are observed events of the glue, and vice versa.

7 The HLA Model

Turning now to the HLA, the core of the Wright formalization is the specification of the RTI connector.[4] At the top-most level the RTI is defined as follows:

Connector RTI($nfeds$: 1..)
Role $\text{Fed}_{1..nfeds}$ = FederateInterface
Glue = RTIBehavior

The RTI is parameterized by the number of federates ($nfeds$) in a joint simulation; there can be an arbitrary number of

[2] Wright also supports the ability to define architectural styles, check for consistency and completeness of architectural configurations, and check for consistent specifications of components. For this paper, however, we will restrict our presentation to just those parts of Wright that concern the specification of the HLA. For further details, the reader is referred to [3].

[3] Wright uses a non-standard interpretation of external choice in the case in which one of the branches is §: specifically, the choice remains external, unlike, for example, the treatment in [18]. See [3] for technical details.

[4] The full Wright specification is about 15 pages long [5]. For the purposes of this paper we present only certain parts of the model to highlight its key features. There are also a few other differences arising from the fact that our final model includes fixes for several of the problems identified in this paper.

them. The behavior of each federate is specified by the role specification FederateInterface. The interface describes the behavior to which a federate must conform in order to participate in the federation.

The specification of the RTI's behavior, on the other hand, is defined by the glue process RTIBehavior. It describes the manner in which the RTI coordinates communication among the federates within an execution. We now examine each of these parts in turn.

7.1 Specifying the Federate Interface

The behavior of FederateInterface is divided into eight parts.

$$
\begin{array}{l}
\textsf{FederateInterface} = \\
\quad \textsf{FedMgmt} \parallel \textsf{DeclMgmt} \parallel \textsf{ObjMgmt} \parallel \textsf{OwnMgmt} \parallel \textsf{TimeMgmt} \\
\qquad \parallel \textsf{DataMgmt} \parallel \textsf{FedJoined} \parallel \textsf{ControlPause} \\
\textbf{where} \\
\quad \textsf{FedMgmt} = \ldots \\
\quad \textsf{DeclMgmt} = \ldots \\
\quad \ldots
\end{array}
$$

Within each part, services are represented as events. A federate-initiated a service like "Join Federation Execution" appears as $\overline{\textsf{joinFedExecution}}$, while a RTI-initiated service like "Initiate Pause" appears as initiatePause. The required and permitted orderings of the events are specified by a process that indicates what events can follow other events, and where choices can be made by the federate or the RTI.

To determine the legal orderings and choice points we relied primarily on the published IFSpec documentation. For example, in Figure 3 the "Request Pause" service (initiated by a federate) would correspond to the $\overline{\textsf{requestPause}}$ event. There are three preconditions for this service. The first two indicate that the $\overline{\textsf{createFedExecution}}$ and $\overline{\textsf{joinFedExecution}}$ must preceed any occurance of that event. The relative ordering of those later two events and their relation to other events in the system must be inferred by looking at other parts of the IFSpec documentation.

The third precondition in the example is more problematic. What exactly does it mean for a federation to be "advancing"? Resolving this kind if issue is trickey because the precondition refers to a state of the RTI and not the API. To handle this kind of situation we had to infer the existence of RTI state and build that into the RTIBehavior process. However, sometimes the informal description was sufficiently vague that we had to go back to the designers of the HLA to ask them what they had in mind. For example, there was no place in the IFSpec where "not paused" was defined, and we had to clarify what the intention was—specifically, what events should be allowed to occur in a "paused" state and which are forbidden. (Note *some* events must be allowed; otherwise there would be no way to "unpause" the federate.)

Structurally, the first six parts of the specification correspond to the six management groups of the IFSpec (cf., Section 3). The last two processes represent relationships among events in different management groups. We structure the description as the parallel composition of subprocesses for two reasons. First, it supports traceability: each of the management processes corresponds to a distinct part of the original IFSpec. When problems are discovered it is relatively easy to trace them to the source. Second, the decomposition permits us to separate concerns. This is fairly obvious for the case of the six management groups, since each covers a distinct aspect of the integration standard. (We will consider these shortly.) Less obviously, however, we can use separate processes to localize both the definition of common constraints, as well as ways in which events in one management group affect what is permissible in another.

Localization of common constraints is illustrated by the FedJoined process:

$$
\begin{array}{l}
\textsf{FedJoined} = \overline{\textsf{joinFedExecution}} \rightarrow (\textsf{RUN}_{FedEvents} \\
\qquad\qquad\quad \triangle\ \overline{\textsf{resignFedExecution}} \rightarrow \S)
\end{array}
$$

The process constrains a federate from invoking any service until it has first joined the federation. Formally, after initiating the joinFedExecution event, a federate can engage in any of the events in the set *FedEvents*. This set includes all HLA events except for federation setup and takedown events. However, once the event resignFedExecution is executed, it interrupts the RUN process (indicated by the CSP interrupt operator, $\triangle$) and leads to successful termination of the federate. By virtue of the way CSP synchronizes events across parallel processes, placing this process in parallel with other processes has the effect of forcing all other parts of the specification to satisfy its constraint.

Because this constraint on invocation of services includes services from all of the management groups, we simplify the specification considerably by putting this constraint into a single process. The alternative would be to include this constraint redundantly in each of separate management group processes, significantly complicating those processes.

Localization of inter-group effects is illustrated by ControlPause:

$$
\begin{array}{l}
\textsf{ControlPause} = \textsf{RUN}_{PauseEvents} \\
\qquad \triangle\ \overline{\textsf{pauseAchieved}} \rightarrow \overline{\textsf{resumeAchieved}} \rightarrow \textsf{ControlPause}
\end{array}
$$

In this process *PauseEvents* is the set of events that should *not* be allowed to occur when a federation is paused. Initially the process permits any of these events to take place. However, when a pauseAchieved event is initiated by a federate, none of those events are permitted until a resumeAchieved event occurs. Since pauseAchieved occurs in response to a pause request that can be initiated by some other federate, and then mediated by the RTI (as we detail later), this links the effects of one federate to other federate behaviors.

Returning to the six management group processes, each such specification describes which services a federate may initiate and under what circumstances. It also describes which services may be invoked on that federate by the RTI, and under what circumstances. To illustrate, Figure 5 details one of the groups, Federation Management.

This extract illustrates how we represent federate behavior and characterize what interactions are possible. Within this specification a key part of the FedMgmt specification is to describe pause and resume behavior of a federate. Referring to Figure 5, we see that after joining the execution, the federate exhibits the behavior described by NormalFedMgmt. That is, it can carry out normal events (like requesting to save or restore state), it is permitted to request a pause, and it should expect the possibility that a pause may be initiated. Once a pause is initiated, it may choose between refusing to pause (and exhibiting the behavior of NormalFedMgmt) and agreeing to pause. If it decides to pause, it then notifies the RTI of its success and exhibits the behavior described in PausedFedMgmt. This is the inverse of NormalFedMgmt with respect to pausing—in this state, it may carry out normal events that are not affected by pausing (which is true of

FedMgmt = JoinFed ⊓ $\overline{\text{createFedExecution}}$ → JoinFed
JoinFed = $\overline{\text{joinFedExecution}}$ → NormalFedMgmt

NormalFedMgmt =
InitiateFedActivity □ WaitForFedActivity ⊓ EndFedMgmt

InitiateFedActivity =
$\overline{\text{requestPause}}$ → NormalFedMgmt
⊓ $\overline{\text{requestFedSave}}$ → NormalFedMgmt
⊓ $\overline{\text{requestRestore}}$ → NormalFedMgmt
EndFedMgmt = $\overline{\text{resignFedExecution}}$ → (§ ⊓
$\overline{\text{destroyFedExecution}}$ → §)

WaitForFedActivity =
initiatePause → (NormalFedMgmt ⊓
$\overline{\text{pauseAchieved}}$ → PausedFedMgmt)
□ initiateFedSave → $\overline{\text{fedSaveBegun}}$ → $\overline{\text{fedSaveComplete}}$ →
NormalFedMgmt
□ initiateRestore → $\overline{\text{restoreComplete}}$ → NormalFedMgmt

PausedFedMgmt =
InitiatePausedFedActivity □ WaitForPausedFedActivity ⊓
EndFedMgmt
InitiatePausedFedActivity =
$\overline{\text{requestResume}}$ → PausedFedMgmt
⊓ $\overline{\text{requestFedSave}}$ → PausedFedMgmt
⊓ $\overline{\text{requestRestore}}$ → PausedFedMgmt
WaitForPausedFedActivity =
initiateResume → $\overline{\text{resumeAchieved}}$ → NormalFedMgmt
□ initiateFedSave → $\overline{\text{fedSaveBegun}}$ → $\overline{\text{fedSaveComplete}}$ →
PausedFedMgmt
□ initiateRestore → $\overline{\text{restoreComplete}}$ →PausedFedMgmt

Figure 5: Specification of FedMgmt

both saving and restoring state), it is permitted to request a resume (but not another pause), and it should expect that an initiateResume will occur. Once it does, the federation returns to it normal behavior.

In the specification of FederateInterface we use non-determinism to abstract away the actual behavior of a specific federate. For example, InitiateFedActivity provides an internal choice among a set of alternatives. The actual choice will depend on the computation of the federate filling the role. Here we simply indicate that one of the possibilities might occur.

7.2 Specifying RTI Behavior

While FederateInterface models the behavior of a single federate, RTIBehavior describes how multiple federates interact via the run-time infrastructure provided by the integration standard. A representative extract of the specification is shown in Figure 6.

Like FederateInterface, the description of RTIBehavior uses multiple processes to separate different aspects of the glue's behavior. As before, these processes can be divided into those encapsulating global constraints and those describing local behaviors.

The global constraints are captured by the two processes HandleMembership and JoinedFeds. These deal (respectively) with how an execution is created and populated, and with keeping track of which federates are currently members of the federation. This information is needed at various times by all of the mini-protocols. By separating out this concern, we simplify each of the mini-protocols, since they need not maintain this state on their own.

RTIBehavior = HandleMembership ∥ $\text{JoinedFeds}_{\{\}}$ ∥
MiniProtocols
where
HandleMembership = ...
JoinedFeds_S =
(whoIsJoined!S → JoinedFeds_S)
□ (□ $i : (1..nfeds)$ • Fed_i.joinFedExecution →
$\text{JoinedFeds}_{S \cup \{i\}}$)
□ (□ $i : (1..nfeds)$ • Fed_i.resignFedExecution →
$\text{JoinedFeds}_{S \setminus \{i\}}$)
□ §

MiniProtocols =
FederationProtocols ∥ DeclarationProtocols ∥ ObjectProtocols ∥
OwnershipProtocols ∥ TimeProtocols ∥ DataDistributionProtocols

FederationProtocols = PauseProtocol ∥ ...
PauseProtocol = HandlePauseResume ∥ $\text{PausedFeds}_{\{\}}$
HandlePauseResume =
(□ $i : (1..nfeds)$ • Fed_i.requestPause →
$\overline{\text{whoIsJoined?S}}$ → $\overline{\text{whoIsPaused?T}}$ →
(; $j : (S \setminus T)$ • $\overline{\text{Fed}_j\text{.initiatePause}}$ → §) ;
HandlePauseResume)
□ (□ $i : (1..nfeds)$ • Fed_i.requestResume → $\overline{\text{whoIsJoined?S}}$ →
$\overline{\text{whoIsPaused?T}}$ → $\text{ResumeResponse}_{S==T,\ T}$)
□ §
$\text{ResumeResponse}_{true,S}$ =
(; $i : S$ • $\overline{\text{Fed}_i\text{.initiateResume}}$ → §) ; HandlePauseResume
$\text{ResumeResponse}_{false,S}$ = HandlePauseResume
PausedFeds_S = ...

ObjectProtocols = HandleRegistrations ∥ HandleRemoves ∥
HandleAttrOutOfScopes ∥ ...

HandleRegistrations =
(□ $i : (1..nfeds)$ • Fed_i.registerObject → $\overline{\text{implicitAOAN!i}}$ →
HandleRegistrations)
□ §

HandleRemoves =
(□ $i : (1..nfeds)$ • Fed_i.deleteObject → $\overline{\text{whoIsJoined?S}}$ →
(; $j : (S \setminus \{i\})$ • $\text{DecideIfRemoveNeeded}_j$) ; HandleRemoves)
□ (□ $i : (1..nfeds)$ • Fed_i.attrsOutOfScope →
$\text{DecideIfRemoveNeeded}_i$; HandleRemoves)
□ (implicitOutOfScope?i → $\text{DecideIfRemoveNeeded}_i$;
HandleRemoves)
□ §
$\text{DecideIfRemoveNeeded}_i$ = § ⊓ $\overline{\text{Fed}_i\text{.removeObject}}$ → §

HandleAttrOutOfScopes =
(□ $i : (1..nfeds)$ • Fed_i.subscribeObjClassAttr →
$\text{DecideImplOutOfScope}_i$; HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.unsubscribeObjClassAttr →
$\text{DecideImplOutOfScope}_i$; HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.subscribeObjClassAttrWithRegion →
$\text{DecideImplOutOfScope}_i$; HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.unsubscribeObjClassAttrWithRegion →
$\text{DecideImplOutOfScope}_i$; HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.publishObjClass →$\overline{\text{whoIsJoined?S}}$ →
(; $j : (S \setminus \{i\})$ • $\text{DecideOutOfScope}_j$) ;
HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.unpublishObjClass →$\overline{\text{whoIsJoined?S}}$ →
(; $j : (S \setminus \{i\})$ • $\text{DecideOutOfScope}_j$) ;
HandleAttrOutOfScopes)
□ (□ $i : (1..nfeds)$ • Fed_i.attrOwnAcqNotification →
$\text{DecideImplOutOfScope}_i$; HandleAttrOutOfScopes)
□ (implicitAOAN?i → $\text{DecideImplOutOfScope}_i$;
HandleAttrOutOfScopes)
□ §
$\text{DecideOutOfScope}_i$ = § ⊓ $\overline{\text{Fed}_i\text{.attrsOutOfScope}}$ → §
$\text{DecideImplOutOfScope}_i$ = § ⊓ $\overline{\text{implicitOutOfScope!i}}$ → §

Figure 6: Specification of RTIBehavior

In the case of JoinedFeds the current membership of the federation is modeled as a set, represented by the subscript (S) of the process. The process communicates the value of this state using the whoIsJoined event. The rest of the definition describes how JoinedFeds monitors the events affecting membership (joinFedExecution and resignFedExeuction) and modifies its state accordingly.

The MiniProtocols process forms the core of the glue. This process is itself a combination of subprocesses, each of which is a mini-protocol defining how the RTI behaves with respect to one aspect of the overall interaction. The mini-protocols are first grouped by management group, for traceability to the IFSpec, and then by service or closely related cluster of services.

We found that it was useful to have two kinds of mini-protocols at the lowest level. The first kind is concerned with specifying the effects of a particular kind of service call. These mini-protocols describe how a request initiated by one federate leads to communication via the RTI with other federates, and how those federates must respond in order for the original request to be fulfilled. For example, the simple HandlePauseResume mini-protocol describes how the RTI reacts to a requestPause event initiated by a federate. In this case it finds out which federates are members of the execution, which federates are already paused, and informs all member federates that are not paused to engage in $\overline{\text{initiatePause}}$. Other mini-protocols, like those handling transfer of ownership, although more complex, are described in a similar fashion.

The second kind of mini-protocol is one that collects *all* the stimuli that can cause a single RTI-initiated service. HandleAttrOutOfScopes is a good example. The RTI is supposed to inform a federate whenever a particular attribute is no longer relevant to that federate. The list of services that could cause this to happen is rather long, and could result in communication from any of the federates, not just the one that will be notified that the attribute is "out-of-scope." Having the stimuli collected in one place makes it much easier to see what causes a given RTI-initiated service.

A key aspect of the specification of the mini-protocols is the use of non-determinism to achieve abstraction. For example, the HandleAttrOutOfScopes mini-protocol collects the stimuli that can cause the $\overline{\text{attrsOutOfScope}}$ event to be invoked by the RTI on a federate. To describe under what conditions an invocation of publishObjClass (one of the stimuli) leads to $\overline{\text{attrsOutOfScope}}$, a lot of information is needed. The decision depends on state that accumulates during the run of the execution (like attribute subscriptions of federates for object classes), as well as the parameters to the triggering service invocation. Instead of representing the precise relationship between two services (like publishObjClass and $\overline{\text{attrsOutOfScope}}$), we simply show that some relationship exists. Looking more closely at HandleAttrOutOfScopes, we notice that each stimulus is followed by a use of the DecideOutOfScope description or the DecideImplOutOfScope description. These descriptions specify that the RTI makes *some* choice about whether or not the stimulus leads to an object attribute going out-of-scope, but does not specify *how* the choice is made.

8 Using the Model

Constructing a formal model for an integration standard as complex as the HLA is a non-trivial task. Many of the ordering relationships embodied in the Wright protocol can be directly inferred from the pre- and post-conditions of services in the original IFSpec. However, as we noted earlier, for many situations, we had to experiment with a number of alternatives, and in many cases get in touch with the designers of the HLA to find out exactly what was the intended behavior. Once it became clear what the behavior should be, the model provided a vehicle for clearly providing a precise definition of it. Indeed, parts of our formal models will be incorporated as supplementary documentation in future releases of the IFSpec.

But the value of the specification goes beyond mere documentation. In the process of formalizing the HLA, we identified several dozen issues that pointed to deeper concerns about the nature of the HLA design—concerns that are crucial to understanding how to use or implement it. Here are two examples:

Exceptions: Each service description in the IFSpec lists a set of exceptions. For example, joinFedExecution has 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). It turned out that the answer was that in some cases exceptions are used to convey important information, while in other cases they represent genuine errors. For example, before a federate can join a federation, the federation must exist. It has the option of creating the federation itself, but there is no way for a federate to determine if this is unnecessary without first attempting to create it, and getting an exception back if it has already been created.

Retained state: To mediate the communication between federates, the RTI must retain certain state. But it is not clear what state, and for how long. For example, when a federate saves its state, it provides a save label. State can be restored through a "restore" service call (using an existing label). But state can only be restored when all federates have a save for the save label being restored. However, in the IFSpec there is no indication of how long this save label can be successfully used: after what point can a federate discard a previous save?

In addition to raising critical issues for clarification, the formal model also helps expose unintended behavior of the standard. We discovered about a dozen such anomalies using a combination of careful review and the facilities of a commercial model checker for CSP, called FDR [8]. To make use of the model checker we used two primary techniques. The first was to look for potential deadlocks in parts of the specification.[5] When the tool detects "deadlock" it provides a trace showing where the process goes awry. Such deadlocks typically indicated the presence of a situation in which different parts of the specification had inconsistent views about the behavior expected of other parts. The other technique was to see if the model was consistent with some desirable behavior. To check for this situation we used to tool to check if a refinement relationship exists between the model and a process that exhibits just that behavior.

The problems that we detected fell into three classes:

[5] In principle one could run the entire model through FDR and find all deadlocks within. In practice, the HLA model is much too large for the checker: so we had to break it into small pieces, and incrementally recombine these in various combinations.

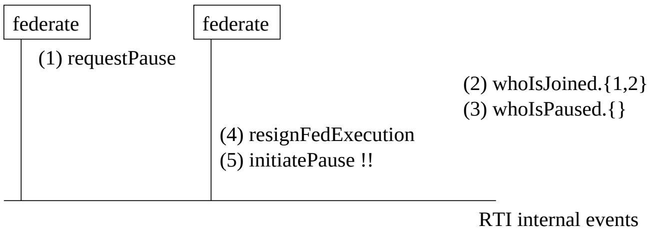

Figure 7: Race condition with resigning federates

Race conditions: Figure 7 shows a trace depicting a race condition we found when analyzing the HLA specification using FDR. The second event, whoIsJoined.{1,2}, depicts the RTI determining the current federation membership. It is does this to inform all federates to initiate a pause (as seen in the HandlePause mini-protocol of Figure 6). However, there is a race condition inherent in this situation. If a federate resigns after the RTI determined membership, the RTI can erroneously attempt to communicate with a federate that is no longer a member of the execution. Had the resignation occurred before the RTI determined federate membership, there would be no problem as the RTI would not attempt to initiate a pause on the resigned federate.

Deadlocks: The next two examples point out different ways in which a federation execution designed to the HLA standard can become deadlocked. Both cases deal with the pause and resume protocols and circumstances under which a federation cannot resume normal execution.

In the first case, we look more closely at the implications of allowing a federate to refuse to pause. Referring back to the WaitForFedActivity description from the FedMgmt process in FederateInterface, we see that after receiving an initiatePause event a federate is allowed to choose either to pause its execution and notify the RTI or to refuse to pause.

Looking next at the HandleResume mini-protocol in the glue specification, we can see the problem with this. The boolean condition to ResumeResponse formalizes a pre-condition to the requestResume service, which states "The federation execution is paused." In order for the federation execution to be paused, each federate that is a member of the federation must be paused. If one federate refuses to pause, the entire federation is not paused and hence normal execution may not be resumed within the execution. Therefore, the ability of a federate to refuse to pause leads directly to the possibility that a federation execution may become deadlocked.

In the second case, we model a potential solution to the first problem by requiring a federate to pause if so directed. However, after adjusting the Wright specification to match this solution, analysis still indicates that the execution may deadlock, but for a different reason. Looking back at the definition of PausedFedMgmt within the FedMgmt process, we notice that a paused federate is allowed to choose whether or not to request a resume. But for this situation FDR generates a trace leading to a deadlock, in which every federate chooses not to request a resume; the federation is deadlocked with every federate "expecting" some other federate to request a resume.

Unlike the first case, however, this example of deadlock does not point to a flaw in the HLA standard. In a number of cases such as this, there may exist several reasonable ways to resolve a problem by suitable choices within a particular federation or RTI implementation. In these cases it would be wrong for the HLA standard to prescribe a particular solution. For example, with respect to the problem of pausing, the standard correctly does not include a requirement that a paused federate *must* request a resume, as there are other legitimate ways in which deadlock can be reasonably avoided. Other policies for ending pauses include designating a particular federate as the one that always must request resumes, or specifying that a time-out should be used to decide when to request a resume.

The real solution is to provide supplementary documentation that highlights such trouble spots and, where possible, indicates possible solutions from which the integrator could select. Our formal model partially serves this role by both identifying the problem areas, and by allowing us to experiment with different policies for resolution.

Unexpected outcomes: The Wright model allowed us to analyze whether certain combinations of behaviors could lead to unintuitive outcomes. Typically, the specification would show immediate behaviors (e.g., that an RTI could reply to a given event in one of several ways), but the question arose, what is the result of composing such behaviors? Are there combinations of choices that lead to unintuitive outcomes?

As an example, consider the following three behaviors: The first immediate behavior is apparent from the HandleRegistrations mini-protocol. Registration is used by a federate to inform the RTI of the existence of a new object, with the result that the registering federate acquires ownership of some (or all) of the object's attributes.[6] The second immediate behavior is apparent from the HandleAttrOutOfScopes mini-protocol: acquiring ownership of an attribute may cause the attribute to go out-of-scope. The third behavior is apparent from the HandleRemoves mini-protocol; when an attribute goes out-of-scope, the federate may be informed that it should remove the object (i.e., that object is no longer relevant to the federate and it should delete its local copy).

When these three behaviors are composed, we can observe that it is possible for the registration of an object to lead directly to the RTI telling the registering federate to remove that object. This, clearly, is not what was intended. However, there remains a question of whether or not the IFSpec does actually allow this chain of activity. Since two of the immediate behaviors were described in mini-protocols that use non-determinism to abstract the real relationships, we still must determine if the composed behavior is possible (i.e., whether this particular sequence of choices is a valid one). By looking back at the IFSpec, we see that this could happen if the registering federate acquires ownership of all the attributes of the object–the composition is possible and there is a problem.

9 Discussion, Conclusion, and Future Work

This paper has described an approach to formalization and analysis of integration standards using the HLA as an example. The effectiveness of this approach is best indicated

[6] Normally, whenever a federate acquires ownership, the attrOwnAcqNotification event is used. In the case of object registration, however, acquiring ownership is part of the post-condition of the registration service and no separate service is used to inform the federate of the ownership change. We explicitly denote such service side-effects that otherwise are noted in separate services by using such events as $\overline{\text{implicitAOAN}}$.

by noting that its identification of issues led directly to significant improvements in the published specification of the HLA. However, in considering the value of the approach it is important to be clear about what is essential, what is incidental, and what still remains to be done.

Among the essential elements we would point to four key techniques. First is the treatment of a component integration standard as a formal architectural model, focusing on the semantics of the connectors as the key issue in need of clarification. Specifically, an architectural approach focuses on the need to model the connection apparatus of the standard, and further helps structure the definition–explicitly separating the interface to the connector (here FederateInterface) from the mediating behavior (here RTIBehavior).

Second is the modeling of that semantics as a protocol. By explicitly representing orders of invocations and loci of nondeterminism and choice, a protocol clarifies many global control and sequencing issues, as well as opening the way for exploration of consequent behavior.

Third is the use of abstraction to make the architectural specification tractable (both intellectually and for model-checking tools). In particular, to do this we abstracted away the details of the data model and decisions of individual federates. While this led to a lack of precision, it greatly simplified the overall specification. (As with any abstraction, however, the downside is that each discovered problem must be carefully examined to determine if its introduction is a consequence of abstracting too far from the actual system.)

Fourth, was a careful attention to structuring the architectural specification. In particular, we divided the 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 provide traceability to the original document.

Among the inessential aspects were the use of CSP and the details of the simulation domain itself. CSP provides the formal basis of Wright, but it is only one of many possible notations that could have been used. As we note below, we found other complementary formalisms to be effective in identifying different kinds of problems. Moreover, other protocol modeling notations would likely have revealed many of the same problems. For example, the developers of Rapide used their event-based architectural modeling language to discover problems similar to those that we identified.

While distributed simulation is unique in some respects, many of the issues that we identify in this paper would apply to virtually any complex integration standard. The HLA is architecturally unusual in so far as it is centered around a single connector (the RTI). Moreover, some of the complexity of the HLA specification comes from the domain of distributed simulation (such as the particular services for time and object management).

On the other hand, we would argue that other aspects of the HLA are embodied by most other integration standards. In particular, most standards must take care to explain how a composition is created, how reconfiguration takes place during run time, how synchronization is handled between multiple components, and what kinds of guarantees are provided for inter-component communication. These aspects are equally pertinent to standards for avionics systems, robotics control systems, and even general-purpose component integration standards such as DCOM or CORBA. And it is these kinds of properties of an integration standard that make it most difficult to understand, implement, and reuse.

It is important to note, however, that formalizations such as ours are just one of many tools and notations. 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 and tools for architectural modeling and analysis.

Indeed, in our own work we we also formalized parts of the specification using StateCharts (which appear in [23]) and Z [7]. In particular, we used these formalisms to handle the state-oriented aspects of the system. For example, a key property that should be maintained by a federation is that there is at most one owner for every simulated object in the system. This property is relatively easy to specify (and check) using a language like Z, but cumbersome using one like CSP.

One important extension of the approach described in this paper is the use of explicit formal models to guide implementors in producing conformant components and run-time infrastructure. There are at least two key conformance issues that a formal model can help resolve. The first is to provide better guidance for implementors. The formal model helps clarify what *must* be included and thereby establishes a baseline for functionality in components and supporting run-time infrastructure. For example, our HLA model clearly indicates that the RTI will have to maintain various kinds of state, including the current list of joined federates, pending pauses, requests for object attribute ownership transfer, etc. Such information is present only implicitly in an API, making it difficult for infrastructure implementors to tell what is essential and what is optional.

A second issue is conformance checking. Given a formal model it should be possible to devise a set of conformance tests that can be used by component and infrastructure implementors. While the generation of tests from formal specifications is itself an active research area, the application of those results to integration standards seems like a particularly promising area for future work.

More generally, the use of formal models for documenting and analyzing integration standards is clearly in its infancy. We will need many more examples, and as noted, broader coverage of other important properties before it becomes clear what are the relative merits of using formal modeling techniques such as those described in this paper. However, we believe the success of the Wright formalization of the HLA is cause for guarded optimism.

10 Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency and Rome Laboratory, USAF, under Cooperative Agreement F30602-97-2-0031, and by the Defense Modeling and Simulation Office (DMSO). 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 Rome Laboratory, the US Department of Defense, or DMSO. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. We would like to acknowledge the help of Richard Weatherly and Reed Little, our main sources of wisdom for the intended behavior of the HLA.

REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[2] R. Allen. Formalism and informalism in architectural style: A case study. In *Proc of the First Intl. Workshop on Architectures for Software Systems*, April 1995.

[3] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, CMU, School of Computer Science, January 1997. CMU/SCS Report CMU-CS-97-144.

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

[5] R. J. Allen, D. Garlan, and J. Ivers. A Wright specification of the HLA. Technical report, Carnegie Mellon University, School of Computer Science, 1998.

[6] E. Clarke et al. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, April 1986.

[7] C. A. Damon, R. Melton, R. J. Allen, E. Bigelow, J. M. Ivers, and D. Garlan. Formalizing a specification for analysis: The HLA ownership properties. Technical Report CMU-CS-98-149, Carnegie Mellon University, School of Computer Science, 1998.

[8] *Failures Divergence Refinement: FDR2 User Manual.* Formal Systems (Europe) Ltd., Oxford, England, version 2.22 edition, October 1997.

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.

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

[11] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[12] 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 Transactions on Software Engineering*, April 1995.

[13] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT LCS, 1988.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings ESEC'95*, September 1995.

[15] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, April 1995.

[16] J. Peterson. Petri nets. *ACM Computing Surveys*, September 1977.

[17] RASSP project overview, Version 1.0. CSIS TR, Dept of Electrical Engineering, University of Virginia, 1994.

[18] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[19] 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 Transactions on Software Engineering*, April 1995.

[20] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Integration of real-time software modules for reconfigurable sensor-based control systems. In *Proc 1992 IEEE/RSJ Intl Conf on Intelligent Robots and Systems.* IEEE Computer Society Press, July 1992.

[21] K. Sullivan, J. Socha, and M. Marchukov. Using formal methods to reason about architectural standards. In *Proceedings of the 1997 International Conference on Software Engineering*, May 1997.

[22] U.S. Department of Defense. High Level Architecture Interface Specification, Version 1.2, August 1997. Also available via http://www.dmso.mil/projects/hla/.

[23] U.S. Department of Defense. High Level Architecture Interface Specification, Version 1.3, draft 1, April 1998. Also available via http://www.dmso.mil/projects/hla/.

A Summary of CSP used in this paper.

We use the following subset of CSP:

- **Processes and Events:** A process describes an entity that can engage in communication events. Events may be primitive or they can have associated data (as in `e?x` and `e!x`, representing input and output of data, respectively).
- **Prefixing:** A process that engages in event `e` and then becomes process `P` is denoted $e \rightarrow P$.
- **Sequencing:** ("sequential composition") A process that behaves like `P` until `P` terminates (§) and then behaves like `Q`, is denoted $P \; ; \; Q$.
- **Interrupting:** A process that behaves like `P` until the occurrence of the first event in `Q`, is denoted $P \bigtriangleup Q$.
- **Alternative:** ("external choice") A process that can behave like `P` or `Q`, where the choice is made by the environment, is denoted $P \mathbin{\Box} Q$. ("Environment" refers to the other processes that interact with the process.)
- **Decision:** ("internal choice") A process that can behave like `P` or `Q`, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Processes may also be subscripted to represent internal state.
- **Parallel Composition:** Processes can be composed using the $\|$ operator. Parallel processes may interact by jointly (synchronously) engaging in events that lie within the intersection of their alphabets. Conversely, if an event e is in the alphabet of processes P_1 and P_2, then P_1 can only engage in the event if P_2 can also do so. That is, the process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2,

In process expressions $\rightarrow$ associates to the right and binds tighter than both $\Box$ and $\sqcap$. So $e \rightarrow f \rightarrow P \mathbin{\Box} g \rightarrow Q$ is equivalent to $(e \rightarrow (f \rightarrow P)) \mathbin{\Box} (g \rightarrow Q)$.