Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

AUTOMATED DISTRIBUTED SYSTEM TESTING: DESIGNING AN RTI VERIFICATION SYSTEM

John Tufarolo
Jeff Nielsen
Susan Symington
Richard Weatherly
Annette Wilson

The MITRE Corporation
1820 Dolley Madison Boulevard
McLean, VA 22102-3481, U.S.A.

James Ivers

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Timothy C. Hyon

TRW International Defense
Simulation Systems
12902 Federal Systems Park Drive
Fairfax, VA 22033, U.S.A.

ABSTRACT

A project is currently underway which involves testing a distributed system – the Run Time Infrastructure (RTI) component of the High Level Architecture (HLA). As part of this effort, a test suite has been designed and implemented to provide a coordinated and automated approach to testing this distributed system. This suite includes the creation and application of a Script Definition Language (SDL) to specify test sequences, and a test executive to control execution of the tests, coordinate the test environment, and record test results. This paper describes the design and implementation of this test environment.

1 INTRODUCTION

In a memorandum from the U.S. Department of Defense (DoD) in 1996, The High Level Architecture (HLA) was established as the standard technical architecture for all DoD simulations (U.S. Department of Defense 1996). The fundamental nature of HLA has been culled from existing technologies and experiences, particularly those found in Distributed Interaction Simulation (DIS) protocols (Voss 1993) and the Aggregate Level Simulation Protocol (ALSP) (Weatherly et. al. 1996). These existing efforts, while serving separate domains, share a common thread of providing a means by which simulations and simulation surrogates can operate together.

Key components of the HLA include:

- a common mechanism for defining and specifying object models (Defense Modeling and Simulation Office 1998b);
- services describing an HLA runtime environment (Defense Modeling and Simulation Office 1998a); and
- rules describing policy with the HLA (Defense Modeling and Simulation Office 1998c).

The second component, known as the *HLA Interface Specification*, describes the functional interface between simulation *federates* (simulation or simulation surrogate participating in a federation) and the runtime environment. The specification also defines a minimal acceptable level of behavior for the runtime environment. The extent of specified behavior is critical since in writing test requirements, only the minimum required behaviors necessitate validation. Behavior beyond the specification permits an RTI developer flexibility and variability with implementation. An implementation complying with the HLA Interface Specification is known as an *HLA Runtime Infrastructure* (RTI). A recent HLA update can be found in (Dahmann et. al. 1998). The interested reader is also referred to (Dahmann et. al. 1997) for additional details on the HLA.

As in many development projects, considerable verification effort is required to ensure faithfulness to a system specification. Such efforts are further complicated when attempting verification of a complex, distributed system (Page et. al. 1997). The RTI exemplifies such a complex system. In many ways, an RTI is similar to a distributed operating system and as such, makes for a challenging verification problem.

This paper describes the design and implementation of an RTI verification system (*The Verifier*) and the experiences therein. Section 2 provides background on the requirements and limitations for testing an RTI, and presents alternatives for the test approach. Section 3 describes the system as it was designed and implemented. Conclusions are included as section 4.

A companion paper (Tufarolo et. al. 1999) describes the application of this system by examining the process of

transforming the HLA interface specification into requirements, test scripts, and presents conclusions and observations.

2 RTI TESTING

The Defense Modeling and Simulation Office (DMSO) sponsored early development of an RTI, both as a proof-of-principle and as a means to evolve the interface specification. Details concerning this initial development of an RTI are described in (Carothers et. al. 1997).

The services in the specification are documented purely from a functional perspective. The standard is very clear about what each service means, what must first be true, and what the results will be. No attempt is made, however, to specify the performance, reliability, or scalability goals (to pick a few examples) of the architecture. By only describing the requisite functionality, the standard has left room for different environments to support the needs of different communities. One RTI implementation for example may run incredibly fast, but won't scale very well to large numbers of federates or object instances. Another RTI implementation may scale to large numbers of federates and object instances at the expense of performance. Regardless, each RTI will conform to the same interface and provide identical functionality.

Because of these specification characteristics, any testing effort must be strictly limited to the functional behavior of an RTI. In designing a verification system, additional scoping was done to help bound the capabilities of the system. Some conclusions were:

- No stress testing will be done; the verifier will not check the limits of how many federates can participate, how many object instances can be created, etc.
- No race conditions will be checked; these are subtle areas that are notoriously hard (and expensive) to reproduce.

2.1 RTI Test Requirements

Testing of an RTI focuses on the requirements and behavior directly evident in the HLA Interface Specification. The pertinent specification (Defense Modeling and Simulation Office 1998a) includes over 200 pages and contains 129 distinct services.

The documentation for each service includes pre- and post-conditions for the service, as well as a prose introduction describing the service and its use. This information allows a reader to determine which sequences of services are permissible, and conditions that should result in exceptions.

Because many of the services are highly correlated, most of test requirements cannot be written by examining each service in isolation. Individual services must be considered in relation to other services with which a relationship exists. Meeting this condition requires that any given test requirement address not only an individual service, but also other inter-related services.

2.2 Test Environment Requirements

In addition to the requirements for testing an RTI, designing an RTI verification system required developing the requirements for the test system itself. These requirements include:

- *Automation of the test execution*

The sheer magnitude of the RTI test requirements established the need for a system that would allow automated test execution with limited intervention.

- *Support for repeated multiple executions*

Because of the anticipated iterative need of the testing, the verification system must easily support repeated (regression) testing.

- *Connectivity to remote processes*

The RTI specifications necessitate the ability for distributed federates to connect to an RTI. As such, the verification system must test this remote connectivity capability.

- *A coordinated view across all distributed processes*

Testing RTI services requires stimulating (by making service calls) separate processes (on one or more machines), and then observing the behavior of each process for the proper responses (or lack thereof). The implication is that the verification system must support a single view of the state of the test execution occurring across the distributed processes. Alternatively the system could interact with the distributed processes individually, then collect the results afterwards.

- *Provide a configurable environment*

The verification system must be able to be tailored to handle a variety of tests, as influenced by the variety of HLA services. For example, one set of tests required interacting with the RTI strictly through a single point of access, while others require interacting with multiple points of access across separate machines concurrently.

- *Allow for evolution of the specification*

The verification system must support update of the test requirements and test specifications to allow continued support for an evolving HLA specification. Tests should be traceable back to a specific item in the specification. The system must also support identifying tests that must be changed in response to specification changes.

- *No special hooks into RTI needed*

The verification should interact with an RTI via the published, code-level, HLA interface without modification. Adhering to this condition allows any RTI to be tested, regardless of whether it's a distributed implementation or a centralized/monolithic implementation.

2.3 Test Approach Alternatives

The task of testing an RTI involves 1) invoking a sequence of federate-initiated services, and 2) observing the RTI's response in the form of RTI-initiated services (federate callbacks). Several techniques to accomplish this were explored when designing the Verifier. For the results of the system to be credible, the correctness of the verification process needed to be as open, reproducible, and simple as possible. This consideration drove the selection of the Verifier approach. Some of the alternative approaches considered are discussed in this section.

2.3.1 Manual Testing

RTI tests could be performed manually. The only software needed to do such testing is the *Test Federate*. This simple program offers a GUI where an operator can invoke federate-initiated services and display the results of RTI-initiated service invocations. The Test Federate software is available at the HLA Software Distribution Center on http://hla.dmso.mil. Manual testing using software like the Test Federate is useful for debugging both RTIs and federations, but the reproducibility required by the verification system would be difficult to achieve.

2.3.2 Use Existing Federations

During the early development of the HLA requirements, an effort was made to create federations that represent all anticipated applications of the HLA. It has been suggested that those same federations could be used to test RTIs. This suggestion is better applied to issues of RTI performance given that each federation represents an interesting pattern of service invocation from which different performance values can be measured. It is not a practical suggestion for the systematic testing of RTI functionality. The constraints of each federate limit the sequences of service invocations available to test the RTI. The desire for openness is not served because the verification process must also include visibility into all federates involved in RTI testing.

2.3.3 New Approaches

Having decided that the Verifier would be a custom-built system rather than formed with real federates, the designers next considered the choice between a distributed or centralized approach. This choice does not refer as much to the disposition of the software components as to how the individual tests are managed.

A distributed Verifier would contain a number of independent service invocation scenarios or scripts. A script is used to control which services are invoked for a given federate. To perform a test, the Verifier selects a set of scripts, creates an instance of a special script interpretation federate for each script, and attaches those federates to the RTI being tested. The exchange of information between each federate and the RTI is recorded and compared to the expected result. If the expected result is obtained, the test is considered successful.

A centralized Verifier would also contain a number of scripts but it would use only a single script for a given test. This script would control all federates involved in the test by selectively reaching out to each federate and specifying which service to invoke. When action is expected from the RTI, the script will listen at one or more federates for the expected RTI initiated service invocation to occur.

Two advantages of the distributed approach are that scripts can be written in any of the programming languages supported by the RTI being tested and operations can be performed on the RTI in parallel. A disadvantage of this approach is the complexity in understanding a given test. To have confidence in the correctness of a test, the behavior of a set of parallel script executions and the result analysis mechanism must be considered as a whole.

An advantage of the centralized approach is that the entire content of a test can be expressed in a single script. Because that script has access to the state of all federates involved with the test, it can react to the RTI and dynamically change the course of the test. This is particularly important given that there is often a large number of correct and incorrect responses to any given test stimulus. A disadvantage to centralized testing is the complexity of the infrastructure needed to connect a single script interpreter to the distributed set of federates at which it will invoke services and receive service invocations. Nevertheless, the cost of constructing the central script interpreter and distributed infrastructure was deemed well worth the clarity and simplicity of the single, centralized script approach and was chosen for the Verifier design.

2.4 Existing Capabilities

William G. Saxon and James F. Leathrum Jr. of the Virginia Modeling, Analysis and Simulation Center (VMASC) at Old Dominion University performed a survey of existing software testing tools (Saxon and Leathrum 1998). The identified tools were divided into two categories: code-dependent and code-independent systems. The code-dependent systems were eliminated as candidates since source code for an RTI under test would not be available.

The code-independent tools included systems such as Ballista, Deploy, ADL (Chang, Richardson, and Sakar, 1995), and CATS. None of these tools fully supported the requirement to provide stimulus at multiple distributed points and observe the response at all points. This led to the development of The Verifier to support this requirement.

Systems such as Ballista have been used to perform robustness testing on earlier RTI implementation (Fernsler and Koopman 1999; DeVale, Koopman, and Guttendorf 1999).

3 DESIGN

The RTI Verifier system architecture consists of a Script Definition Language (SDL) created to specify test scripts; an application executive, controller, and SDL interpreter to parse and execute scripts; test federates to connect and interact with an RTI under test; and a database to maintain requirements, tests, and test results. Figure 1 presents a functional overview of this architecture.

Using this system begins with the HLA Interface Specification. A person must read and understand this specification and distill a set of test requirements for entry into the system. These test requirements form the basis for generating test scripts. Using the v1.3 Interface Specification (Defense Modeling and Simulation Office 1998a), this activity yielded over 1,600 individual test requirements.

Based upon an individual test requirement, one or more scripts are created using the SDL. These scripts are stored in the database and associated with a corresponding test requirement. A Microsoft Access database application was written to provide a convenient method for entering and maintaining test requirements as well as test scripts.

At execution time, a test executive is initiated that manages access to the test scripts, and provides up to five separate *attachment points* (APs) to the RTI under test. The APs connect to *test federates* which in turn, are connected to the RTI under test. A *test controller* GUI allows a human tester to select, initiate, and monitor ongoing test activities. It also permits tests to be hierarchically grouped to allow large or small sets of test cases to be executed. Scripts stored in the database are retrieved via JDBC *(*Patel and Moss 1997). Results from script execution are also stored in the database, again using JDBC as the access mechanism.

Within the Access DB application, test results can be reviewed on-line or in a test report detailing the tests executed, and in the case of an unsuccessful execution, the failure status and the trace of events leading to the failure. Script failures are examined to determine if the failure is a result of behavior contrary to the specification as opposed to an error in *statement* of the test requirement, *implementation* of the script testing the requirement, or a possible alternative interpretation of the specification.

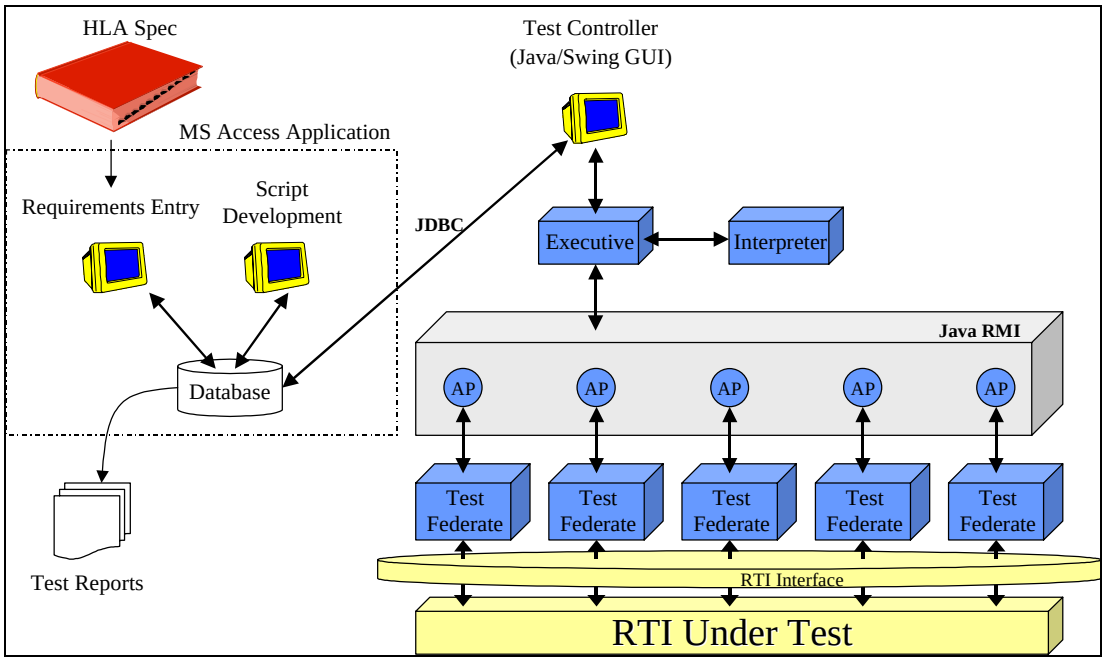

Figure 1: Verifier System Architecture

The following subsections provide more detail concerning components of the verification system architecture.

3.1 Script Definition Language

SDL provides the cornerstone for the verification system. Because the HLA specification includes a unique set of services to test, a convenient mechanism was needed to facilitate generating test cases. A new language – SDL – was devised to address the unique nature of interacting with an RTI.

The following capabilities are fundamental to the SDL:

- Allow invocation of all Federate-initiated services
- Allow acceptance of all RTI-initiated services ("callbacks")
- Allow interaction with the RTI via separate attachment points
- Maintain a consolidated view across all attachment points
- Support HLA data types
- Provide flow of control
- Provide exit status (success or failure)

The basic flow of a SDL script is to:

1. Initiate one or more RTI service calls
2. Look for actions
3. Look for lack of actions
4. Trap unexpected actions
5. Decide what to do next

Figure 2 offers a sample of the SDL.

```
invoke ap1 RTI_SERVICE (arguments)
return (arguments)
  federateNotExecutionMember (theException) ;
  exception (theException) {
   exitFailure ("Service call exception");
  }
accept all
  FEDERATE_SERVICE_1 ap2 (arguments) ;
  FEDERATE_SERVICE_2 ap3 (arguments) ;
  other theService theAP {
   exitFailure ("Unexpected callback");
  }
  unsatisfied {
   println ("Callback not received");
   exitFailure ("Unsatisfied callback");
  }
exitSuccess ("Test completed");
```

Figure 2: SDL Sample

RTI services are initiated by using the ***invoke*** statement, with "AP" identifying the attachment point from which the service call will be initiated. Within the context of a single script, RTI calls can be sent to multiple attachment points concurrently. Based on the particular RTI service used, the arguments will contain the necessary data for the service. Likewise, the optional ***return*** statement will capture any return values that are associated with the service call.

Federate services (callbacks from the RTI) are captured via the ***accept*** statement. In these services, the "AP" identifies the attachment point at which the federate service callback is expected. More than one federate service callback can be specified in the context of a single accept statement. The ***all*** keyword requires receipt of every federate service listed for the accept statement to be considered satisfied. Alternatively, the keyword ***any*** can be used to indicate that only one of the federate services listed is needed to satisfy the statement. Unexpected callbacks are captured via the ***other*** keyword. The ***unsatisfied*** keyword allows the script to stop waiting for unfulfilled callbacks from the RTI after a fixed waiting period. Because the test suite is not intended to test performance or race conditions, this waiting time is usually large (the default value is 5 seconds) and can be specified at the test controller GUI.

Failures during script execution can emerge as a result of RTI exceptions, script language exceptions, and behavioral errors. If an exception occurs while using the ***invoke*** statement, it can be captured by including the ***exception*** keyword. A decision to continue or abort the script can be contained in the exception handling code. For some test cases, specific exceptions are expected. These exceptions can be included by name in addition to the general exception case.

The exit condition from a script is specified via the ***exitFailure*** or ***exitSuccess*** keywords. Based on the requirements of the test, these exit statements are placed in the appropriate locations. It is incumbent upon a script designer to use these statements properly to signal the ultimate success or failure of the script. An exception to this is the case of a failure in the script. A script can "fail" when a callback is received and not captured, an exception is thrown and not caught, or because of a semantic script error. All of these occurrences will automatically result in the script terminating as if the *exitFailure* mechanism was used.

Other details of the language include variable type definitions, print statements, comments, program logic flow control, etc. These are typical of other programming languages and the details are not included here.

3.2 SDL Interpreter

To implement the use of the SDL in this system, a parser was designed and created to read and interpret test

statements written in SDL and in turn to interact with an RTI via the distributed test points. In order to minimize development efforts, existing products were sought to support building this parser and integrating into the other Java-based components of the system. JJTree and JavaCC were selected to meet these criteria.

JavaCC is a Java language preprocessor that generates top-down (recursive descent) parsers based upon a detailed grammar (Sun Microsystems 1999a). By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic look-ahead to resolve shift-shift ambiguities locally at these points (i.e., the parser is LL(k) only at such points, but remains LL(1) everywhere else for better performance). Shift-reduce and reduce-reduce conflicts are not an issue for top-down parsers (Barrett et. al. 1986). Top-down parsers have other advantages (as opposed to more general grammars) such as being easier to debug, having the ability to parse to any non-terminal in the grammar, and having the ability to pass values (attributes) both up and down the parse tree during parsing.

JJTree is an add-on to JavaCC, allowing the generated parser to produce syntax trees, i.e., it inserts tree-building actions at various places in the JavaCC source (Sun Microsystems 1999b). By default, JJTree generates code to construct parse tree nodes for each non-terminal in the language. This behavior can be modified so that some non-terminals do not have nodes generated, or so that a node is generated for a part of a production's expansion. JJTree defines a Java interface node that all parse tree nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them.

The SDL parser was built by creating a *context-sensitive* grammar. This grammar specification was used by JJTree to produce inputs for JavaCC. The result was Java code, which implemented the specifications of the grammar for use by the test executive.

3.3 Test Executive

To execute the scripts, the SDL interpreter works together with the test executive. This process serves as the central point of control during RTI testing. It is responsible for both accessing the scripts and managing their execution. Scripts can be either read from a file or (more commonly) retrieved from the database.

In the database, scripts are organized into a hierarchy that supports four different levels of execution. A *script* can be executed either by itself or as part of a sequence of scripts that comprise a *test*. In the latter use, scripts serve as components, which are combined in different ways to form a coherent test. (For example, a script that creates a federation execution and joins three federates can be re-used as the beginning of any number of different tests.) Likewise, a *series* is a group of one or more tests to be run together, and a *scenario* is a group of series.

The test executive supports execution at any one of these four levels of execution and ensures that scripts are executed and/or aborted as appropriate. As it executes a script, test, series, or scenario, it records the results in the database.

3.4 Test Controller GUI

The test controller GUI is a graphical front end to the test executive that serves as the point of access for an RTI tester. From the GUI, a tester can observe and control the behavior of up to five different federates connected to the RTI under test. Each of the test executive's five attachment points can be linked by the user to a remote test federate process. The GUI allows the tester to view the contents of the database and to select scripts, tests, series, or scenarios for execution. The user can then monitor the progress of an execution by watching the display.

The GUI also permits the user to manually invoke any of the federate-initiated services and respond to RTI-initiated services at any of the attachment points. This can be done at any time, whether or not an execution is in progress.

The test controller, test executive, and SDL interpreter are all written in Java. These can thus be run on any platform for which there is a Java virtual machine. The GUI elements were written using the Java "Swing" classes, which are now part of the standard Java 1.2 class libraries. These are a set of graphical components that provides a consistent and platform-independent look and feel, so that the GUI appears and behaves the same way no matter what it is run on.

3.5 Test Federates

An RTI can not be tested unless there are federates joined to it that are using its services. This is the role of the test federate processes that are at the heart of the testing system. A test federate is a completely user-driven federate that has the capability to exercise the full set of federate-initiated services and accept the full set of RTI-initiated services ("callbacks"). An example is the test federate provided with the DMSO RTI, which allows the user to call any RTI service by selecting it from a menu and supplying the appropriate parameters in a dialog box.

The test federate program written for the RTI verification system is different in that it provides no direct user interface. Instead, all of the running test federate processes are controlled completely by the test executive. The executive tells the test federate which RTI services to call, and any callbacks that arrive must be processed and "cleared" by the executive before the federate proceeds.

Again, this is so that the executive can maintain centralized control over the system at all times.

The test federates are also written in Java, and use the Java Native Interface to call the pre-compiled C++ RTI libraries.

3.6 Remote Connectivity

Connectivity between the executive/interpreter and the test federate processes is achieved through the Java Remote Method Invocation (RMI) protocol. The RMI architecture provides a way for objects in one Java process to invoke methods on objects from other Java processes—either local or remote (Sun Microsystems 1999c). (This is in the same spirit as the OMG CORBA architecture, but is specific to Java programs.) An RMI registry running on each machine provides a simple naming service, and the marshalling and unmarshalling of parameters and return values is handled transparently (via object serialization) from the programmer's point of view. Clients simply obtain a reference to a remote object and then call its methods in exactly the same way as with local objects.

Thus, from the point of view of the test executive, each remote test federate is simply a "test ambassador" object that provides methods mirroring all of the RTI service calls. To invoke a federate-initiated service on the RTI under test, the test executive simply calls the appropriate method on the test ambassador. Everything else, connecting to the remote test federate process, sending data over the network, etc., is handled automatically.

In a similar fashion, each test federate has a reference to a single "controller" object, to which it passes all RTI callbacks. The test federate does not concern itself with the fact that the controller is actually on a remote machine. It simply calls the appropriate method and waits for it to return.

3.7 Database Application and Connectivity

Connectivity between the test executive and the database is achieved with the Java Database Connectivity (JDBC) API. JDBC allows Java programs to execute SQL statements on any relational database using a single, standard Java interface (Patel and Moss 1997). SQL statements are used both to query the database (retrieving scripts, tests, etc.) and to update it (when recording execution data). The particular implementation of the JDBC used in the verifier software also takes advantage of Java RMI.

4 CONCLUSIONS

The complexities in testing a distributed system such as an RTI pose a considerable challenge, even after limiting the testing to functional capabilities and disregarding performance, stress-testing, and race conditions. Particular test environment requirements, including a need to stimulate and observe the response of an RTI at multiple distributed points, drove the development of a new system, *The Verifier*, to test an RTI. The Verifier consists of a Script Definition Language to specify test scripts; an application executive controller and SDL interpreter to parse and execute scripts; test federates to connect and interact with an RTI under test; and a database to maintain requirements, tests, and test results.

Application of this system to date has proved to be very useful to meet the unique needs of RTI testing. A companion paper (Tufarolo et. al. 1999) provides more details into the process of using The Verifier. Future efforts include enhancing the script definition language and test environment as needed to address the next version of the HLA standard (IEEE Standard "P1516.1 Draft Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification").

ACKNOWLEDGEMENTS

The authors would like to acknowledge the efforts of Reed Little of the Software Engineering Institute (SEI) at Carnegie Mellon University and Dave Seidel of the MITRE Corporation for their assistance in creating and reviewing this paper. The U.S. Department of Defense (DoD) Defense Modeling and Simulation Office (DMSO) supports the work on the Verifier Project.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

REFERENCES

Barrett, W., Bates, R., Gustafson, D., and Couch, J. 1986. *Compiler Construction Theory and Practice*, Second Edition, Chicago, IL: Science Research Associates, Inc.

Carothers, C.D., Fujimoto, R.M., Weatherly, R.M., and Wilson, A.L. 1997. Design and Implementation of HLA Time Management in the RTI Version F.0, In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Androdottir, K.J. Healy, D.H. Withers, and B.L. Nelson, 373-380. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Chang, J., Richardson, D.J., and Sankar S. 1995. *Automated Test Selection from ADL Specifications*, in First California Software Symposium (CSS'95), March.

Dahmann, J.S., Fujimoto, R.M., and Weatherly, R.M. 1997. The Department of Defense High Level Architecture, In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Androdottir, K.J. Healy,

D.H. Withers, and B.L. Nelson, 142-149. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Dahmann, J.S., Fujimoto, R.M., and Weatherly, R.M. 1998. The DoD High Level Architecture: An Update, In *Proceedings of the 1998 Winter Simulation Conference*, ed. D.J. Medeiros, E.F. Watson, J.S. Carson, and M.S. Manivannan, 797-804. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Defense Modeling and Simulation Office. 1998a. High Level Architecture Interface Specification, v1.3.

Defense Modeling and Simulation Office. 1998b. High Level Architecture Object Model Template, v1.3.

Defense Modeling and Simulation Office. 1998c. High Level Architecture Rules, v1.3.

DeVale, J., Koopman, P., and Guttendorf, D. 1999. The Ballista Software Robustness Testing Service. To appear in *Testing Computer Software '99*, June.

Fernsler, K., and Koopman, P. 1999. Ballista Robustness Testing of the HLA RTI v.1.0.3 Preliminary Report. Carnegie Mellon University Report, at http://www.ices.cmu.edu/ballista/reports/hla_rti_1_0_3.pdf

Page, E., J. Tufarolo and B. Canova. 1997. A Case Study of Verfication, Validation and Accreditation for Advanced Distributed Simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(3):393-424.

Patel, Pratik, and Moss, Karl. 1997. *Java Database Programming with JDBC, 2nd Edition*, Coriolis Group Books, NY.

Saxon, W.G., and Leathrum, J.F. 1998. Survery of testing tools, Informal Briefing.

Sun Microsystems Incorporated. 1999a. Java Compiler Compiler – The Java Parser Generator, at http://www.suntest.com/JavaCC

Sun Microsystems Incorporated. 1999b. JJTree Introduction, at http://www.suntest.com/JavaCC/DOC/JJTree.html

Sun Microsystems Incoporated. 1999c. Java Remote Method Invocation (RMI), at http://java.sun.com/products/jdk/rmi

Tufarolo, J., Nielsen J., Symington, S., Weatherly, R., Wilson, A., Ivers, J., and Hyon, T. 1999. Automated Distributed System Testing: Application of an RTI Verification System, *1999 Winter Simulation Conference* ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, and G.W. Evans. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

U.S. Department of Defense. 1996. DoD High Level Architecture (HLA) for Simulations. U.S. Departement of Defense. Memorandum signed by USD(A&T).

Voss, L.D. 1993. *A Revolution in Simulation: Distributed Interaction in the 90's and Beyond*, Arlington, VA: Pasha Publications, Inc.

Weatherly, R.M., Wilson, A.L., Canova, B.S., Page, E.H., Zabek, A.A., and Fischer, M.C. 1996. *Proceedings of the 29th Havaii International Conference on Systems Sciences*, Volume 1 (Wailea, HI, Jan. 3-6), 407-415.

AUTHOR BIOGRAPHIES

JOHN A. TUFAROLO is a Lead Simulation Systems Engineer for the MITRE Corporation in Reston, Virginia, where he is currently involved in High Level Architecture (HLA) testing and HLA federation development activities. Mr. Tufarolo is the Information Director for the Association of Computing Machinery (ACM) Special Interest Group on Simulation (SIGSIM), and a member of the ACM, IEEE CS, and SIGSIM. His professional interests include discrete event simulation, simulation systems development, and military modeling and simulation. Mr. Tufarolo has a BS degree in Electrical Engineering from Drexel University and an MS in Systems Engineering from George Mason University.

TIMOTHY C. HYON is a Software Developer for the TRW Systems & Information Technology Group in Fairfax, Virginia, where he is currently developing an advanced simulation system known as the Planning Support Function (PSF). PSF will provide a software tool to help Japan's new system of centralized government better prepare itself to handle natural disasters, humanitarian assistance, and national defense. Prior to joining TRW, he was a Senior Simulation and Modeling engineer for the MITRE Corporation in Reston, Virginia, where he was a lead engineer for the HLA Runtime Infrastructure (RTI) and RTI Verifier system development projects. Mr. Hyon received a BS degree in Electrical Engineering from the University of Delaware and an MS in Electrical Engineering from Georgia Institute of Technology.

JAMES IVERS is a member of the technical staff at the Software Engineering Institute where he works in the Architecture Trade-off Analysis initiative. He is interested in formal methods and analysis, particularly as applied to software architectures. He received a BA in Computer Science and Mathematics from Transylvania University and an MSE in Software Engineering from Carnegie Mellon University.

JEFF NIELSEN is a Senior Software Systems Engineer at the MITRE Corporation. His current HLA-related activities include providing technical and management support to DMSO-sponsored federation efforts, participating in the HLA IEEE specification development, and developing the

RTI Verifier test suite. Mr. Nielsen holds an MS in Computer Science and a M.A.Ed. in Instructional Technology, and is currently pursuing a Ph.D. in Computer Science. He is a member of IEEE.

SUSAN SYMINGTON is a Lead Scientist at the MITRE Corporation where she is involved in HLA testing activities using the Verifier. She is also the chair of the IEEE High Level Architecture Working Group that drafted the three HLA draft standards: P1516, P1516.1, and P1516.2. She holds a BA in Mathematics and Philosophy from Yale University and an MS in Computer Science from the University of Maryland at College Park.

RICHARD WEATHERLY is the Chief Engineer of the MITRE Corporation's Information Systems and Technology division where he leads their DoD High Level Architecture (HLA) Runtime Infrastructure (RTI) software development team. He is a core member of the Defense Modeling and Simulation Office (DMSO) Technical Support Team and contributor to their HLA Interface Specification, Time Management, and Data Distribution Management working groups. He received his Ph.D. in Electrical Engineering from Clemson University. Please see http://ms.ie.org/weatherly for recent work.

ANNETTE L. WILSON is a Lead Modeling and Simulation Engineer in the Information Systems and Technology Division at the MITRE Corporation, McLean, VA. She is currently project engineer for the RTI Verifier, is a member of the RTI 1.3 development team, and supports the CADRE program by providing RTI expertise for participating federations. Ms. Wilson was one of the developers of the ALSP Infrastructure software (AIS) and chaired the AIS subgroup of the ALSP Interface Working Group. Ms. Wilson has a BS in Computer Science from Texas A&M University and is pursuing an MS in Computer Science at George Mason University.