Evolution of the Standard Simulation Architecture

Dr. Jeffrey S. Steinman
Chief Scientist, RAM Laboratories
10525 Vista Sorrento Parkway, Suite 220
San Diego, CA 92121

steinman@ramlabs.com

Douglas R. Hardy
Scientist, SPAWAR Systems Center-San Diego
53140 Systems Street, Code 244201
San Diego, CA 92152

douglas.hardy@navy.mil

Abstract

This paper proposes the standardization of a layered simulation architecture that addresses the critical modeling needs of the DoD simulation community. The Standard Simulation Architecture works with HLA to provide the additional infrastructure necessary for developing highly inter-acting, decoupled software models, while simultaneously supporting technology infusion from R&D organizations.

A layered architecture is proposed to modularize critical capabilities including high-speed communications between nodes in a multiprocessing federate, general-purpose software utilities, modeling semantics, time management, interest management, and automated interoperability with HLA. The interface layers must be standardized to promote (1) model development, (2) portability and interoperability with other models, (3) scalable high performance, and (4) technology infusion from the research community. Through the standardization process, COTS, GOTS, and Open Source business models are supported.

The Standard Simulation Architecture extends interoperability and reuse principles to (1) the entities residing within a multiprocessing federate and to (2) the components hierarchically residing within an entity or within components. This standardized hierarchical modeling paradigm promotes development of reusable entity and component repositories that can be reused to support different modeling applications. Instead of providing only course-grained interoperability through HLA, the Standard Simulation Architecture also supports medium and fine-grained interoperability between entities and their components.

1 Introduction

Software development efforts funded by the Department of Defense must be regarded as important long-term investments. Complex software systems should be leveraged and reused in other programs whenever technically feasible. In order for this to occur, software must be developed from the start with the goal of reuse. Basic interoperability principles should be carefully followed for software to be successfully reused in other programs. The Standard Simulation Architecture (SSA) promotes these principles with the goal of maximizing the return on software investments that were funded by taxpayer dollars.

The High Level Architecture (HLA) was successfully launched in 1996 to promote interoperability and reuse between simulations executing in distributed environments. These simulations, called federates, typically interoperate in an HLA federation to support joint analysis or joint training exercises that require the coordination of disjoint models. Despite the success of HLA, similar techniques have not yet been fully standardized for promoting interoperability between (1) entities, potentially executing on multiprocessor computers, and (2) their internal components. There are no general-purpose entity or component repositories in existence today. Furthermore, a common architecture has not yet been formally standardized to minimize the development cost required to build software models that integrate with HLA.

Without such standards, it is nearly impossible to develop reusable entity and component software models because they typically embed various critical framework or simulation engine services within the code to coordinate their activity. Because each simulation typically provides its own event-processing engine with its own specialized interfaces, interoperability between entities and components is not possible. Examples of this may include (1) event-scheduling interfaces, (2) interest management with automated data distribution, and (3) time

management services. The Standard Simulation Architecture defines high-performance modeling constructs along with a suite of programming utilities to simplify model development for software engineers while simultaneously minimizing the dependencies between models.

One of the goals of the Standard Simulation Architecture is to maximize flexibility and composability. *Federations* are composed of *simulations*, which are composed of *entities*, which are further hierarchically composed of *components*. The Standard Simulation Architecture promotes flexibility in efficiently mapping the software models to machines operating in a parallel and distributed environment.

Three levels of granularity naturally arise within the Standard Simulation Architecture. *First*, the High Level Architecture normally provides network-based communications between simulations with overheads that are typically in the millisecond range. *Second*, the Standard Simulation Architecture provides high-speed communication between entities potentially executing on multiprocessor machines through shared memory or high-speed network communications. Overheads between interacting entities are typically in the microsecond range. *Third*, components hierarchically modeled within an entity interact with other components through abstract polymorphic function calls with overheads in the nanosecond range.

Roughly six orders of magnitude separate these three levels of granularity. Special consideration must be given to all three levels of granularity to maximize overall performance when designing large parallel and distributed reusable simulation systems. Portability and composition flexibility are most critical when the target hardware platforms vary in different operational settings.

The Standard Simulation Architecture is defined by a dependency-layered approach. Each software layer provides a standard set of interfaces and depends only on the preceding (or lower) layers of the architecture. By standardizing each layer, it becomes possible for technologists to integrate successful R&D efforts into mainstream simulation programs. The proposed Standard Simulation Architecture supports COTS, GOTS, and Open Source business models, thereby providing a way for commercial, government, and academic institutions to participate in developing simulation technology with the necessary infrastructure for promoting interoperability and reuse.

This paper proposes a layered architecture for supporting software interoperability and reuse in DoD simulations where scalability and efficient run-time performance is crucial. The proposed architecture is derived from lessons learned in support of the Joint Simulation System (JSIMS) program, and other large-scale modeling and simulation efforts. Most of the capabilities described in this paper have been developed and successfully used on various government programs. This paper attempts to bring these technologies together into a coherent standardized architecture. Related technologies feeding into the Standard Simulation Architecture include:

1. The High Level Architecture (HLA)
2. The Aggregate Level Simulation Protocol (ALSP)
3. Distributed Interactive Simulation (DIS)
4. Semi-Automated Forces (MODSAF, JSAF, ONESAF, etc.)
5. Common Object Request Broker Architecture (CORBA)
6. Publish/Subscribe architectures
7. Active Routing
8. Time Warp Operating System (TWOS)
9. Synchronous Environment for Emulation and Discrete Event Simulation (SPEEDES)
10. The Joint Simulation System (JSIMS) Common Component Simulation Engine (CCSE)
11. WarpIV high-performance parallel and distributed simulation kernel

12. High Performance Computing Run Time Infrastructure (HPC-RTI)
13. Mixed Resolution Modeling Aide (MRMAide)

First, the historical evolution of the SSA is provided. Then, a general discussion on the subject of interoperability and software reuse shows how familiar concepts taken from HLA can be reused within a simulation framework to promote interoperability, not just between federates, but also between entities and their components. A layered *straw-man* architecture is then presented that supports the full set of interoperability and performance requirements for DoD simulations. These proposed layers are primarily based on operational software that has been developed and reused across a number of programs. Finally, a high-level strategy for developing the *Standard Simulation Architecture* is detailed in the conclusion of this paper.

2 Historical Evolution of the SSA

The evolution of the SSA began in the late 1980's and is still evolving today through the development of core infrastructures for several large DoD projects including the Joint Simulation System (JSIMS).

In the late 1980's, SIMNET was developed to support real-time battlefield simulations of tanks in a virtual training environment. The Joint Training Confederation (JTC) was developed to integrate models from the different armed forces to support joint training exercises. Meanwhile, the Time Warp Operating System (TWOS) was developed at the Jet Propulsion Laboratory (JPL) showing that optimistic time management could achieve parallel speedup when applied to military simulation applications.

In the early 1990's, SIMNET evolved into the Distributed Interactive Simulation (DIS) standard to support virtual battles involving Semi-Automated Forces. IEEE standardized more than one hundred Protocol Data Units (PDUs) that specify message formats exchanged between DIS models. The Aggregate Level Simulation Protocol (ALSP) was developed by MITRE to simplify the integration of various simulations participating in the JTC. Meanwhile, the Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) operating system was developed at JPL as a next-generation high-performance simulation engine to replace TWOS. SPEEDES introduced new flow control techniques that were required to stabilize run-time performance for optimistic simulations.

In the late 1990's, HLA became the interoperability standard for building Federations out of real-time and/or logical-time simulations. As HLA was maturing, the Standard Modeling Framework (SMF) and an initial implementation of the DSMS layer were being designed and developed in SPEEDES. These capabilities have been further enhanced in the WarpIV simulation kernel developed by RAM Laboratories, Inc.

In early 2000, JSIMS combined SPEEDES and HLA as its simulation architecture. This enabled each JSIMS Development Agent (DA) to develop independent models that would interoperate using a powerful SPEEDES-based Common Component Simulation Engine (CCSE), developed at SPAWAR Systems Center (SSC). A new implementation of the DSMS layer was required to support the modeling needs of large complex federations involving multiple SPEEDES and direct HLA Federates. The SPEEDES-HLA combination, with extensions in WarpIV, is currently evolving into the Standard Simulation Architecture (see

Figure 1).

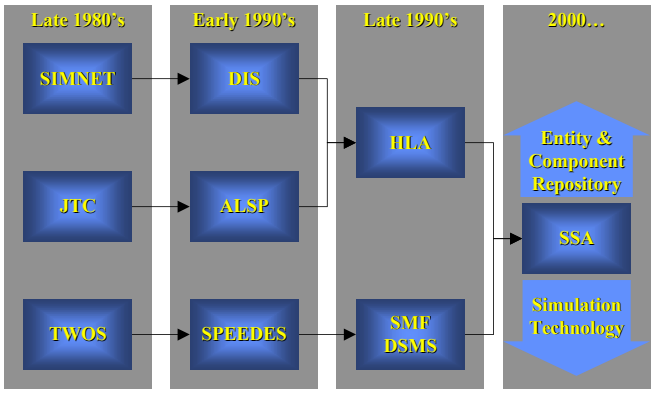


Figure 1: Evolution of the Standard Simulation Architecture from the late 1980's through 2000 and beyond.

3 General Principles of Software Reuse

Software reuse is an important goal to strive for in the area of software engineering. Object-oriented approaches to software development have shown promise in providing interoperability and reuse. However in practice, this goal has been elusive and at best difficult to achieve. Either the *not-invented-here* syndrome overrides the potential for software reuse, or the available software choices presented to engineers simply *do not meet the requirements* of a new software project.

Often, the effort to reuse existing software is larger than what it would take to simply write new code. *Software reuse can require a steep learning curve*. Many times, there are good technical reasons to not reuse software. For example, it might make more sense to redundantly develop new software to *avoid code dependencies*. Under the right circumstances, it may be better to optimize an internal algorithm rather than to reuse a more generic algorithm that would result in *worse performance*. Sometimes, a tight *delivery schedule* dictates the software reuse policy.

Experienced software engineers intuitively use common sense when making decisions concerning software reuse. However, there are several overarching principles that promote the development of reusable software.

1. Reusable software must be passive and not active in its usage. It must not be directly tied to the particular application or infrastructure that is using the software. This means that the services provided by reusable software must be capable of being invoked by any application, not just one in particular. Global variables tied to specific applications should not be accessed by reusable software.[1]
2. A framework is often required to coordinate the operation of reusable software. Typically the coordination is accomplished through run-time dynamic binding mechanisms (e.g., virtual functions, callbacks, polymorphic methods, event scheduling, etc.). These software modules depend on the existence of the framework to coordinate their operation. While frameworks are often necessary to support complex systems, this often limits potentially reusable software to only operate within the framework.[2]
3. Software frameworks can be standardized with a well-defined API to reduce dependencies on any particular vendor's implementation of the framework.[3] This extends interoperability and reuse for both application software and the technology or infrastructure necessary to support the applications.
4. Generic types, generic algorithms, macros, compile flags, abstract base classes, and operator-overloading techniques are tools that can be used to successfully decouple

[1] A good example of reusable software is the Standard Template Library (STL) in C++ that provides a set of general-purpose container classes and generic algorithms.

[2] A good example of a standard framework is how event scheduling and processing depends on the existence of a simulation engine that provides an API to schedule and process events. Even though these events may be reused in several different simulation applications, they still depend on the event scheduling and event processing API of that particular simulation engine. A standardized simulation engine API would allow the events to be reused in other simulation engines that adhere to the same standard.

[3] The High-Level Architecture (HLA) provides a standard API that allows vendors to implement their own Run-Time Infrastructure (RTI). HLA-compliant simulations can interoperate with other HLA-compliant simulations using any HLA-compliant RTI because the interfaces for the RTI have been standardized.

software from a particular application or framework.[4]

5. Software reuse strategies must take communications, computations, memory usage, and overall software engineering complexity into consideration. All too often, complex systems are designed by naively connecting *black-box* software modules together without regard for performance and scalability. The performance of a poorly thought-out system may be disastrous when only connectivity is considered in the design.[5] Even worse is when the software becomes overly brittle through tight coupling between its internal components to the point where it cannot be maintained.
6. Software that was not designed for reuse will almost certainly not be reusable. It takes a disciplined effort to make software reusable.[6]
7. Software cannot claim to be reusable unless it is used in more than one application. One way to help enforce this rule is to test the software in an environment that is isolated from the primary application. Library dependencies should be verified when testing reusable software. It is not uncommon for inappropriate software dependencies to creep into the code-base by *quick fixes* that occur over the life cycle of a software project, especially when employee turnover on the project is high.

4 HLA Reuse Principles Applied to SSA

Within the DoD simulation arena, the discussion of interoperability and reuse has centered on HLA. Four important interoperability principles have emerged from the development of HLA that are directly applied to the SSA.

1. A standardized software *framework* with well-defined interfaces is required to interconnect reusable models (e.g., the RTI).
2. The *data* exchanged by the models must follow an agreed upon standard (e.g., the FOM).[7]
3. *Distributed object technology* allows models to (1) know about each other's state and (2) invoke actions within other models in a coordinated manner (e.g., TM, OM, DM, DDM, and OWN).
4. The *double-abstraction barrier principle* allows a model to invoke actions on other models while hiding the details concerning which specific models are participating in the action and which methods those participating models provide to handle the action (e.g., Interactions).

The Standard Simulation Architecture (SSA) applies these four principles through its conceptual hierarchical decomposition of interoperable software models. This is shown in Figure 2. HLA Federations are composed of two kinds of Federates, SSA Federations and Legacy Federates.[8] SSA Federations are composed of SSA Federates and High Performance Computing (HPC) RTI Federates.[9] SSA Federates are

[4] To illustrate the *generic type* technique, a "smart" pointer class can be defined within a memory management system to provide automated checks for memory problems. If implemented carefully, the smart pointer class can be redefined at compile time to be a normal pointer. This permits reusable applications to operate normally without requiring inclusion of the memory management framework

[5] For example, the overhead in passing data through a network is orders of magnitude larger than the overhead involved when passing data through shared memory or through function calls.

[6] A related corollary to this is, "If it hasn't been tested, then it doesn't work!"

[7] One of the most difficult problems facing the HLA user community is (1) how to specify what goes into the FOM, and (2) how that maps to a particular Federate's SOM. A Federate that participates in more than one Federation must be able to translate its SOM to multiple FOMs. Tools have been developed to accomplish this, but a standardized FOM, such as the RPR-FOM, would significantly help support interoperability.

[8] A Legacy Federate provides its own simulation engine and internal infrastructure to communicate directly with the RTI.

[9] HPC-RTI Federates are also legacy federates, but their interface to the RTI is directly provided by the Standard Simulation Architecture. This provides a more direct interface with lower overheads on high-performance multiprocessor computers.

composed of entities that are hierarchically composed of components. Both entities and components can create/publish local Federation Objects (FOs) and subscribe to remote Federation Objects. Filters may be dynamically created and/or changed to determine which FOs are discovered by which entities and components.

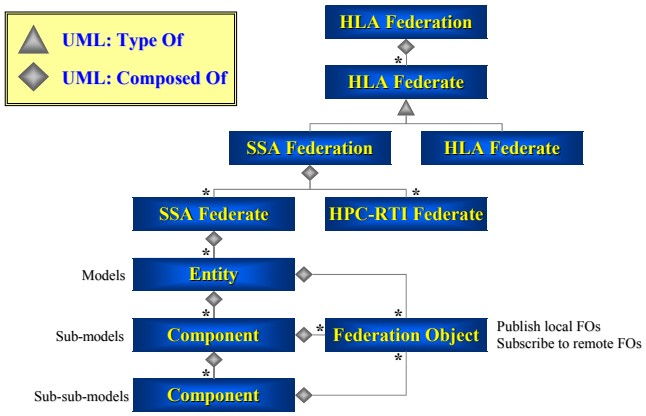


Figure 2: Decomposition of an HLA Federation into Federates, Entities, Components, and Federation Objects.

4.1 Standardized Software Framework

First, while HLA provides a Run Time Infrastructure (RTI) that allows Federates to interoperate, a larger, more comprehensive, *framework* is needed to support the full modeling needs required by a Federate. A Federate's connection to the RTI is only one layer in the overall Standard Simulation Architecture. Opportunity for interoperability and reuse is only realized if the architecture layers are standardized and if the modeling services provided by the framework promotes basic software interoperability and reuse principles.

4.2 Standard Description of Exchanged Data

Second, HLA provides a standard Object Model Template (OMT) format to describe the *data* that flows through the RTI between Federates. A more comprehensive description of the exchanged data between reusable software models representing entities and their components within an SSA Federate is needed to facilitate interoperability and reuse in an efficient manner. A standardized interface between interoperable software modules within a Federate must be defined in a manner that promotes interoperability and reuse for a broad number of simulation applications. Data translations and polymorphic abstractions can help integrate models of mixed-resolution without requiring strong coupling between models. This approach provides a roadmap for transitioning legacy model components into SSA federates.

4.3 Distributed Object Technology

Third, HLA provides *distributed object technology* between Federates. Similar mechanisms must also be provided between entities within an SSA Federate executing on multiple processors. An automated Distributed Simulation Management Services (DSMS) layer within a Federate should mirror HLA functionality between entities while preserving the abstraction that an entity could reside within the Federate or within another Federate. This means that entities within a Federate should learn about each other's state through DSMS Federation Objects (FOs) and interact with one another through DSMS Interactions.[10] An HLA gateway coordinates DSMS activity with the RTI to automate connectivity with other Federates in an HLA Federation.

An entity should never directly call a method on another entity since interacting entities could reside within different Federates. In a similar vein, high-performance Federates executing in parallel must follow the same FO and Interaction guidelines for entities because entities may reside on different processing nodes. This important rule even applies to entities residing on the same processing node in a parallel Federate because the entities themselves may be at different logical times.[11] Like HLA, high performance and scalability is achieved through the DSMS layer by

[10] In many ways, entities within a Federate look like miniature Federates. Entities interoperate exclusively through the exchange of FOs and Interactions supported by the DSMS layer.

[11] Both optimistic and conservative time management schemes use a *scheduler* to determine which entity gets to process its next event. In optimistic simulations, straggler messages can roll entities backwards in time, while other entities continue to process forward. Schedulers in conservative simulations often use topology knowledge between entities to determine which entity gets to process its next event. In general, entities may be at different logical times.

supporting scalable interest management services that throttle FO and Interaction data exchanges between entities and their internal components.

4.4 Double-abstraction Barrier Principle

Fourth, HLA supports the *double-abstraction barrier principle* between Federates through the Interaction mechanism. Federates first subscribe to the kinds of interactions they need and then establish their own methods for handling interactions as they are received. A Federate sending an interaction through the RTI does not know which other Federates (if any) have subscribed to the interaction. This is the first abstraction barrier. Furthermore, even if the sending Federate were to know which other Federates received the interaction, the sending Federate still does not know which methods are applied by receiving Federates to process the interaction. This is the second abstraction barrier.

In a similar manner, the double-abstraction barrier principle can be applied to entities within a Federate and to their internal components. This technique decouples entities and components, thereby promoting reusability in a manner that is familiar to HLA. The difference, however, is that the SSA simplifies the process through its tailored modeling constructs and programming interfaces.

Network-based Federations apply the double-abstraction barrier principle through interactions with typical network overheads in the *millisecond* range. Entities residing within a sequential or parallel Federate interact with one another through DSMS Interactions with much smaller overheads that are typically in the *microsecond* range. Shared memory, rather than network-based communication protocols, provides several orders of magnitude faster communication between entities executing in parallel on high-performance multiprocessor machines. Of course, entities in sequential Federates interact through event-scheduling function calls with slightly lower overheads.

Hierarchical components managed within an entity can interact with one another through the use of polymorphic functions and methods that again preserve the double-abstraction barrier principle. Like callback systems, a component can invoke a polymorphic function that in turn activates polymorphic methods that were registered by objects in other components. This is very similar to general-purpose GUI callback systems that allow applications to register handlers when buttons are pushed, etc., except that the polymorphic method system is fully object-oriented. The hierarchical polymorphic method mechanism also provides scope resolution to restrict which methods in the component hierarchy are activated. Polymorphic methods are invoked through function calls with typical overheads in the *nanosecond* range.

The hierarchical component infrastructure within an entity manages which methods have been registered by which components. The invoker of a polymorphic function does not know which components have registered polymorphic methods, nor does the invoker know which polymorphic methods are applied by registering components when activated. Thus, the double-abstraction barrier principle is maintained.

The polymorphic method system is much more powerful than the standard object-oriented inheritance and virtual function approach to polymorphism. It does not require inheritance or virtual functions to achieve polymorphism. Instead, a special macro is used to define the polymorphic interface (see Code Segment 1).[12]

Code Segment 1: Macro interface for defining a polymorphic interface with *N* arguments. In the current implementation, *N* can range from 0 to 20 to support up to 20 arguments in the generated interface.

```
DEFINE_POLYMORPHIC_INTERFACE_<N>_ARGS(
  FunctionName, ArgType1, ArgType2, …, ArgTypeN
)
```

This macro generates a new polymorphic function that can be invoked to activate

[12] Code examples of interfaces for polymorphic methods are taken from the SPEEDES-based JSIMS Common Component Simulation Engine.

corresponding polymorphic methods that have been registered (see Code Segment 2). Note that the invoker of the polymorphic function only references the interface and has no knowledge of how the interface triggers software models in other components.

Code Segment 2: Macro-generated functions for invoking registered polymorphic methods. When invoked, the first version of the function activates all registered polymorphic methods within the entity. The second version of the function provides scope resolution within a component and its children components to only invoke those registered methods in the specified component substructure.

```
void FunctionName(
  ArgType1, ArgType2, …, ArgTypeN
)

void FunctionName(
  Component *, ArgType1, ArgType2, …,
ArgTypeN
)
```

Through another macro, any number of objects may define one or more of their methods to correspond to the polymorphic interface. Registered objects requires no special inheritance, their object class name can be arbitrary, and their registered method name can also be arbitrary. An example of this is shown in Code Segment 3.

Code Segment 3: Defining a polymorphic method on a class. Notice that the name of the class and its polymorphic method can be anything. The last argument in the macro, *N*, is the number of arguments in the interface. No inheritance is required when defining polymorphic methods.

```
class ClassName {
  private:
  protected:
  public:
    void MethodName(
      ArgType1, ArgType2, …, ArgTypeN
    );
};

DEFINE_POLYMORPHIC_METHOD(
  FunctionName, ClassName, MethodName, N
)
```

The DEFINE_POLYMORPHIC_METHOD macro generates new and uniquely named functions that can be used to register or unregister the polymorphic method within an entity or within any component in an entity's component hierarchy. The interface for registering and unregistering polymorphic methods is shown in Code Segment 4.

Code Segment 4: Macro-generated functions for registering and unregistering polymorphic methods associated with a class. Note that the first two interfaces register and unregister the method with the entity. The second two interfaces register and unregister the method with a component.

```
void
REGISTER_FunctionName_MethodName(
  Entity *, ClassName *
)

void
UNREGISTER_FunctionName_MethodName(
  Entity *, ClassName *
)

void
REGISTER_FunctionName_MethodName(
  Component *, ClassName *
)

void
UNREGISTER_FunctionName_MethodName(
  Component *, ClassName *
}
```

An example of how components work with polymorphic methods is shown pictorially in Figure 3 with an extended UML class diagram.[13] In this example, a radar component on a ship entity sends detections to the track fusion component through the polymorphic method mechanism.

[13] The extended UML diagrams use different shapes and colors for boxes to represent different kinds of classes within a simulation. This helps to quickly convey information without cluttering the diagram. For example, framework objects are drawn as blue boxes, a user-derived simulation object is drawn as a red box, an event is drawn as a red circle, a process is drawn as a red ellipse, polymorphic methods are drawn as purple hexagonal polygons, Arrows indicate scheduling information, etc.

The radar component generates detections that are processed by the track fusion component when invoking the *ProcessDetections* polymorphic function. This in turn activates the *FuseDetections* method in the track fusion component that has been registered as a polymorphic *ProcessDetections* method. The double-abstraction barrier principle is demonstrated in this example to show that the radar component does not know about the track fusion component, nor does it know the name of the track fusion component's method that is applied when processing the detections.

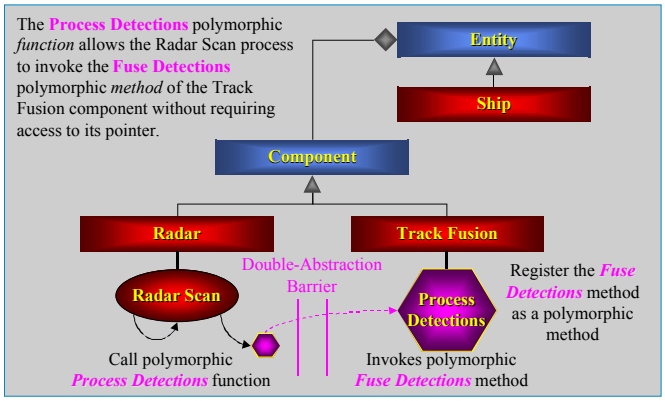


Figure 3: An example of two components interacting through a polymorphic method. A Ship entity contains a Radar component and a Track Fusion component. The Track Fusion component registers its *FuseDetections* method as a *ProcessDetections* polymorphic method. The Radar Scan process periodically feeds its detections to the Track Fusion component by calling the *ProcessDetections* polymorphic function. The Radar does not know that the Track Fusion component exists, nor does it know that the FuseDetections method was registered as a polymorphic method for ProcessDetections. This preserves the double-abstraction barrier principle.

In summary, HLA interoperability and reuse principles can be applied within the proposed Standard Simulation Architecture to address three distinct levels of granularity:

1. **Federates** within an HLA Federation
2. **Entities** within a parallel or sequential Federate
3. **Components** hierarchically composed within an entity

HLA provides a standard framework (RTI) to facilitate distributed object technology in a Federation. The data exchanged between Federates (i.e., Federation Objects and Interactions) is defined in the FOM. The double-abstraction barrier principle is achieved through HLA Interactions.

A high-performance simulation framework can facilitate distributed object technology between entities on parallel high-performance computers. The data exchanged between entities is done in the same manner as HLA using a Distributed Simulation Management Services layer to coordinate the distribution of Federation Objects and Interactions between entities. The double-abstraction barrier principle between entities is again achieved through Interactions.

Working with the Distributed Simulation Management Services layer, an HLA gateway maintains the important abstraction that entities could reside within any Federate or on any node within a Federate executing in parallel.

A standardized hierarchical component framework allows entities to be hierarchically decomposed into sub-models. A macro is used to define polymorphic interfaces that specify the data exchanged between components within an entity. The double-abstraction barrier principle is achieved through the polymorphic method infrastructure.

The Standard Simulation Architecture specifies a framework that can support all of these interoperability principles through a layered approach. Through API standardization, these layers can be developed in an open environment by commercial organizations, government laboratories, and academic institutions.

5 High-Level Modeling Concept

The Standard Simulation Architecture promotes high-speed interoperability and reuse at three different levels. First, Federates can interoperate through HLA interfaces using the HPC-RTI, or they can interoperate directly within the Standard Simulation Architecture. All HLA federates (including the SSA Federation) interoperate through a well-defined FOM and

through standard usage of the RTI.[14] An example of eleven interoperating Federates is shown in Figure 4.

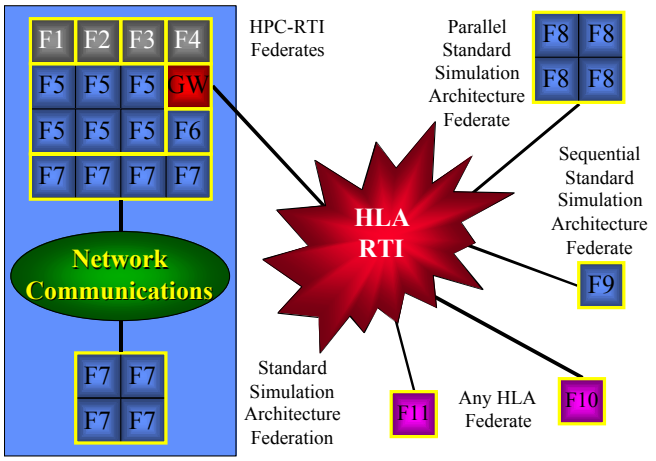


Figure 4: Interoperability between Federates in the Standard Simulation Architecture. In this example, F1, F2, F3, and F4 are HPC-RTI Federates executing on four nodes of a sixteen node parallel machine. F5 is a six node Federate, F6 is a sequential Federate, and F7 uses four nodes on each of the two machines within the group of Standard Simulation Architecture Federates. Communication within a Standard Simulation Architecture Federation is provided through shared memory and/or network message passing. A gateway connects these seven Federates to a network-based RTI to provide interoperability with other HLA Standard Simulation Architecture Federates executing either in parallel (F8) or sequentially (F9), and with legacy Federates (F10 and F11).

Second, entities within the Standard Simulation Architecture interoperate in parallel through the DSMS layer. This means that entities obtain information about other entities by subscribing to each other's published Federation Objects. Entities process events scheduled by other entities using the formal DSMS Interaction mechanism.[15] This not only supports the parallel processing paradigm, but also maintains the important abstraction that interacting entities could reside within different HLA Federates.

The HLA gateway preserves this important abstraction by seamlessly providing connectivity between HLA Federates.[16] If required, time management is coordinated using conservative and/or optimistic techniques between and within each Federate.

Object clustering techniques allocate entities to specific nodes within each Federate. This alleviates the need for all of the software within a federate being linked into one monolithic executable.

An example showing how interactions are sent by one entity and then received by subscribing entities is depicted in Figure 5. The important abstraction that the entities could reside in any Federate is preserved.

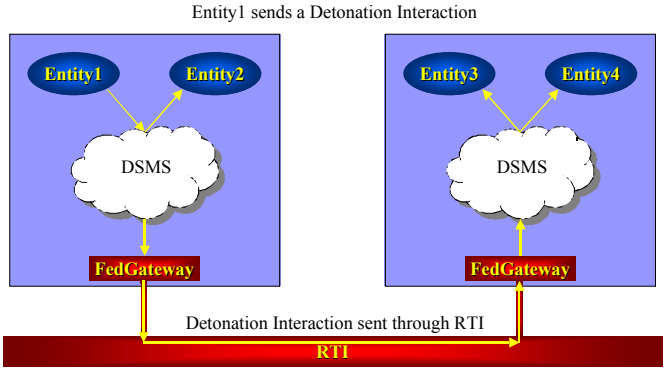


Figure 5: An example of an entity sending an interaction. At time 100, Entity1 uses the DSMS layer to send an interaction scheduled for time 200. Assuming that the lookahead through the RTI is 10, the FedGateway receives the interaction at time 190. It then schedules the interaction through the RTI for time 200. At time 200, all subscribing entities in both Federates receive and process the interaction. The receiving entities have no special dependencies concerning which Federate sent the interaction. This preserves the abstraction that any entity could reside in any Federate.

Third, components within entities interoperate through fully specified type-checked interfaces using polymorphic functions and methods.[17] Components decompose models hierarchically

[14] Because HLA does not define standards for initialization procedures, synchronization points, representation of time, object models, and representation of data, it may be difficult to identify standard usage of the RTI. Eventually, further standards need to be created to truly establish a greater degree of interoperability between HLA federates.

[15] Interactions can be directed to a list of specified entities using a special parameter in the parameter set to store their unique identifiers. The interaction will only be sent through the gateway to other federates if any of the recipient entities are not located within the federate. Significant overhead reductions are possible with this approach.

[16] The HLA Gateway is actually implemented as an entity that subscribes to Federation Objects and Interactions through the DSMS layer and with the RTI.

[17] Components can also abstractly interact with each other by directly invoking the ProcessInteraction method. While this approach has additional parameter packing and unpacking overheads, and provides less type-checking in its parameter set interface, it can sometimes be helpful in reusing abstract interaction handlers. However, event-scheduling overheads are eliminated by directly invoking the ProcessInteraction method.

within an entity to support arbitrary levels of fidelity and detail. Like entities, components also coordinate the publication and subscription of Federation Objects and interactions with interest management. This is shown with a UML class diagram in Figure 6.

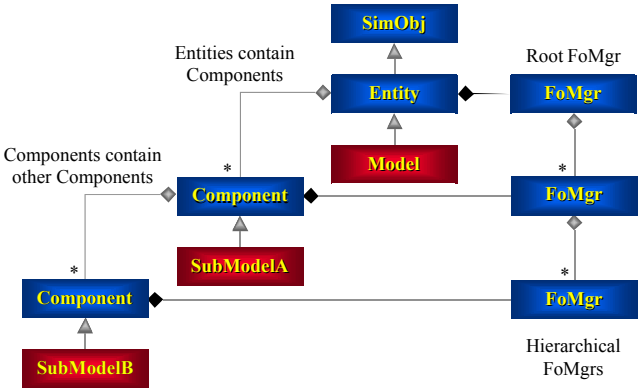
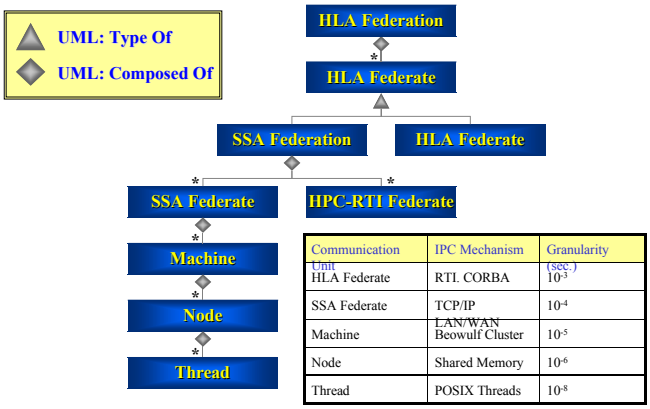


Figure 6: Entities, components, and FoMgrs. Hierarchical components are used to decompose an entity model into sub-models. The components connect their Federation Object Managers (FoMgrs) in the same hierarchical manner to provide efficient interest management between components within an entity. A Federation Object that is discovered by an entity will be directed to the components within the entity as specified by their interest management filters.

From a different perspective, another way to visualize the different levels of granularity within the architecture is to consider Inter Process Communication (IPC) mechanisms. The UML Diagram in Figure 7 shows the hierarchical decomposition of an HLA Federation (see also Figure 4) as it relates to the different levels of IPC granularity.



Communication Unit	IPC Mechanism	Granularity (sec.)
HLA Federate	RTI. CORBA	10^{-3}
SSA Federate	TCP/IP	10^{-4}
Machine	LAN/WAN Beowulf Cluster	10^{-5}
Node	Shared Memory	10^{-6}
Thread	POSIX Threads	10^{-8}

Figure 7: A UML diagram showing the hierarchical decomposition of an HLA Federation in the Standard Simulation Architecture. An HLA Federation is composed of one or more HLA Federates that take on two flavors, Legacy Federates and SSA Federations. The SSA Federation is composed of one or more SSA Federates and/or HPC-RTI Federates. SSA Federates can be spread over one or more Machine. Each Machine contains one or more Nodes. Each Node contains one or more Threads.

At the finest level of granularity, threads allow multiple lightweight processes to communicate within a single heavyweight process (or node). These lightweight processes coordinate through mutual exclusion locking mechanisms that safeguard memory accesses.

Multiple nodes may communicate on a multiprocessor machine. Multiple nodes on a machine normally communicate through high-speed shared memory. However, multiple machines may connect through a local area network to form a Standard Simulation Architecture Federate. These machines typically communicate through standard network protocols such as TCP/IP.

Standard Simulation Architecture Federates may connect together through shared memory, local area networks, and/or wide area networks to form an SSA Federation, which behaves as a single HLA Federate because it has one connection to the RTI through its gateway.

At the lowest level, HLA Federates typically communicate through standard Internet protocols such as TCP/IP, UDP/IP, and IP-multicast. These Internet protocols may be further abstracted using distributed object communication mechanisms

such as CORBA. [18] The RTI provides another layer of abstraction to support federates communicating in a distributed environment. Multiple HLA Federates can connect together to form an HLA Federation.

The coordination of HLA time management and interest management services may add additional overheads to the basic message-passing overheads. Typical levels of granularity for each kind of communicating units are summarized in Figure 7.

Using the HPC-RTI interface, legacy simulations in the Standard Simulation Architecture benefit from high performance parallel processing in three ways. *First*, legacy simulations can self-Federate through the HPC-RTI to execute in parallel on multiprocessor machines. For example, entities could be distributed to four nodes on a multiprocessor machine, thereby reducing the computational load required by a single CPU (see Figure 8).

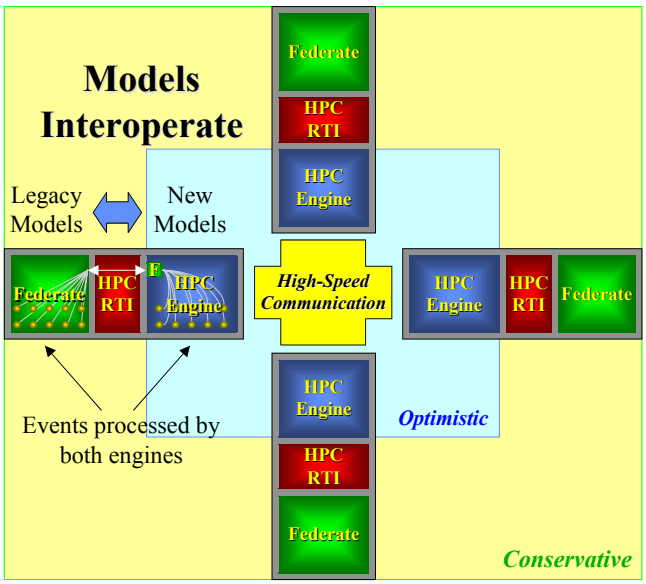


Figure 8: An example of interoperability and high-performance computing through the HPC-RTI on four nodes of a multiprocessor machine. Entities are distributed to multiple instances of the legacy Federate to achieve parallel processing by the Federate. The internal HPC Simulation Engine can also provide models that interoperate with each other and with the Federate. Optimistic computing may occur within the HPC Engine, but through advanced time management techniques, the Federate only receives valid data from the HPC Engine.

Second, the HPC-RTI actually connects two simulation engines together within a single process. It does this in a way that supports interoperability between models implemented in the two engines without sacrificing performance. Integrating a legacy federate with reusable entities or components that are modeled in the Standard Simulation Architecture can extend the functionality and software lifetime of legacy simulations.

Third, Federates using the HPC-RTI can participate in Standard Simulation Architecture Federations executing on high-performance computers (see Figure 4). One very important capability provided by the HPC-RTI over traditional RTIs is that everything, including DM, DDM, and OWM services are potentially managed in logical time. Furthermore, the HPC-RTI provides a seamless integration between mixed real-time and logical-time modes of operation.[19]

[18] The first RTI developed by DMSO used the ORBIX implementation of CORBA. The more recent RTI-NG implementation is based on the ACE/TAU object request broker.

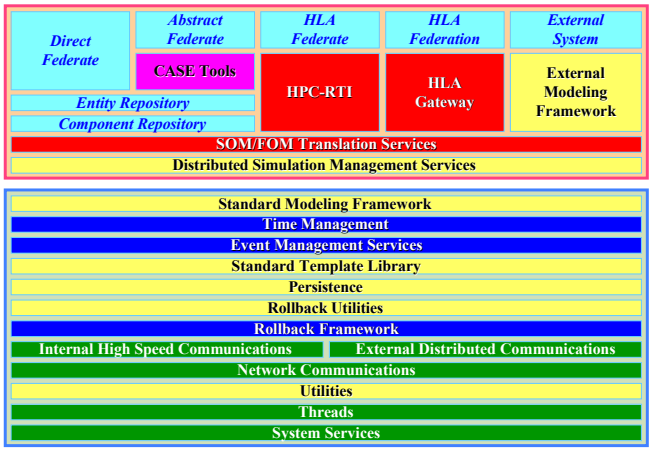
6 The Standard Simulation Architecture

This section first provides an overview of the layered architecture. It then provides a detailed description of each layer in the architecture.

6.1 Architecture Overview

The proposed *Standard Simulation Architecture* is shown in Table 1. It is comprised of multiple software layers that simulation systems build upon.[20] The standardization process defines the set of interfaces for each of these layers.[21] Once this is accomplished, different implementations of these layers can be combined to form complete simulation infrastructures that may be optimized for different types of simulations, communication networks, computing platforms, operating systems, languages, and compilers. Commercial organizations (Commercial Off The Shelf - COTS), government laboratories (Government Off The Shelf - GOTS), and academic institutions (Open Source) can independently contribute their own optimized implementations of any of these layers. A quick overview of the SSA layers is provided below.

Table 1: The Standard Simulation Architecture. Notice that the color-coding of the layers indicates groupings of related functionality. Green layers indicate low-level interprocess communications or system services. Dark blue layers indicate frameworks that coordinate processing. Yellow layers indicate programming interfaces for model developers. Red layers provide HLA services. Light blue layers indicate models or federate applications. Finally, the purple layer indicates graphical tools that can be used to simplify model construction, scenario generation, and data analysis.



The System Services, Threads, Network Communications, Internal High-Speed Communications and External Distributed Communications layers provide a full-spectrum of system utilities and inter-process communication services in a standard portable manner.

The Rollback Framework, Event Management Services, and Time Management layers provide the basic infrastructure that is necessary to support discrete-event and real-time simulations executing on single or multiple CPU machines.

The Utilities, Rollback Utilities, Persistence, Standard Template Library, Standard Modeling Framework, Distributed Simulation Management Services, and External Modeling Framework layers provide the basic set of constructs and tools required for software developers to efficiently build simulation models and to directly connect them to external systems such as graphical user interfaces and hardware devices. Persistence is critical for supporting checkpoint/restart and dynamic load balancing functionality.

The SOM/FOM Translation Services, HPC-RTI Interface, and HLA Gateway layers support interoperability between Standard Simulation Architecture Federates, legacy HLA Federates,

[19] Because the HPC-RTI layered on top of a parallel and distributed time-managed simulation engine, all operations are fundamentally coordinated in logical time. To support real-time federates, events are simply time-tagged by the wall clock. This unified approach to managing events does not require separating real-time messages from logical time messages in message queues.

[20] The layered architecture selectively allows upper layers to invoke services provided by lower layers, not just the layer immediately below. In this way, the layered architecture is like public inheritance in object-oriented software.

[21] One of the first issues to address when standardizing interfaces is programming and naming conventions. This includes standards for naming classes, data members, methods, functions, arguments, local variables, macros, and macro-generated functions. It also involves standards for global variables, error handling, header file rules, operator overloading, and templates. The programming standards do not include rules for bracket indentation, or other personal style issues that can be resolved through *pretty print* formatting tools.

and HLA Federations. The FO and Interaction data translation services allow a Federate to define its own specialized SOM while promoting interoperability with other Federates. The HLA Gateway may provide multi-level security services between networked Federates.

The Component Repository and the Entity Repository provide a library of models that were designed for reuse across multiple simulation domains. Note that entities interoperate through federation objects and interactions, while components interoperate through polymorphic methods.

The CASE tool layer allows commercial vendors to generate code through specialized compilers and/or graphical programming environments to simplify the development of new models. The CASE tool layer may also provide graphical tools to simplify scenario generation and object compositions with mixed levels of resolution. The CASE Tool layer may also provide backward compatibility services to map legacy simulations to the standard simulation Architecture.

The complete architecture provides high-speed software reuse and interoperability between SSA federates, entities, and components. It further provides interoperability with legacy Federates and HLA Federations through the HPC-RTI Interface and HLA Gateway layers. Non-HLA external systems such as high-speed hardware or specialized graphical displays may integrate and directly interoperate with the overall system through the External Modeling Framework.

6.2 Architecture Layers

This section provides further descriptions of each layer in the architecture. Note that each layer at most only depends on the layers below in the architecture.

6.2.1 System Services

In order to preserve portability between operating systems, the System Services layer abstracts all of the system-specific services that might be invoked by the Standard Simulation Architecture. Examples of these services include operations such as forking a process, spawning the execution of a new program, obtaining the time of day, determining CPU usage, waking up the process when a message arrives, establishing network connections, creating/deleting shared memory segments, etc.

6.2.2 Threads

The Threads layer defines portable standard interfaces for supporting lightweight processes across different operating systems and languages.[22] The thread interfaces must be implementable for both UNIX (e.g., Pthreads, Solaris threads, DCE threads, etc.) and Windows operating systems (e.g., WIN32 Threads). This layer must minimally support C++ and Java programming languages. The interface must also include default functions for systems that do not support multithreading.[23] Basic services include the following.

1. Ability to spawn and terminate a thread.
2. Ability to assign a priority to a thread.
3. Mutual exclusion mechanisms.
4. Storage of local data associated with a thread.
5. Method to provide the maximum number of threads.
6. Method to provide the number of active threads.

6.2.3 Utilities

The Utilities layer defines a standard set of interfaces for general-purpose classes including

[22] Threads, or lightweight processes, allow an application to have more than one process active within a single heavyweight process. They share the full memory state of the application; so all memory is "shared memory" in a multithreaded process. This is not the same as running multiple heavyweight processes that communicate through specially created shared memory segments. Threads can run concurrently on machines with multiple CPUs, which can often provide parallel speedup.

[23] A system that does not support multithreading sets the maximum number of threads equal to one.

random number generation,[24] data parsers, XML parsers, various container classes, dynamic arrays, strings,[25] generic algorithms, timers, math utilities, motion libraries,[26] data logging, portable big/little endian data types, I/O stream extensions, memory management tools, object factories, checksum, data compression algorithms, error handling, and internal memory tracking. When applicable, these utilities should be thread-safe, which is why they depend on the threads layer.

6.2.4 Network Communications

The Network Communications layer defines the interfaces for the standard communication infrastructure that is used to connect networked simulations together. A general-purpose client/server infrastructure coordinates message passing between machines in a local area network and between multiple local area networks in a wide area network.[27] Standards such as CORBA may be used to support this layer. However, it is important to define interfaces that are powerful, yet open to the research community. An over-reliance on commercial products may not support innovative R&D efforts that explore new protocols and performance optimizations. Possibly, a simplified version of the CORBA interface is needed.

A Publish/Subscribe wide-area network approach efficiently distributes data between platforms by evaluating subscription filters on the published data, or meta-data associated with each outgoing message. Multiple servers may be used to connect local networks to other local networks. In this manner, spider-web networks of servers try to minimize message congestion while optimally routing messages. This approach supports the usage of reliable message passing services while still conserving bandwidth consumption. An example of this is shown in Figure 9.

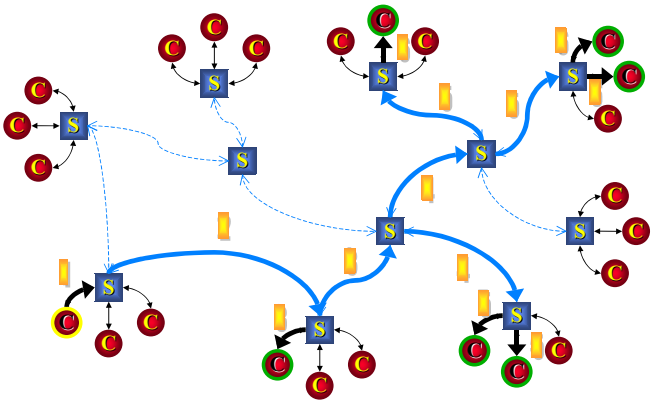


Figure 9: An example of a distributed spider-web network where one publisher sends a message to multiple subscribers. The message is provided to the subscribers as it is routed once through the network. Notice that this active routing approach minimizes bandwidth consumption. 31 hops are required if the message is individually sent to each destination. The smarter active routing approach in this example only requires 13 bandwidth-consuming hops.

The network layer must support dynamic connectivity to allow new applications or routers to join the system and fault tolerance when applications exit the system. The client/server infrastructure must be distributable, provide redundant routing paths, facilitate multiple network protocols, and coordinate multiple application groups when more than one application shares the network. It must also support heterogeneous networks that mix big and little endian data formats. Additional network services to fulfill security requirements may be implemented in this layer.[28]

A generic client/server model is used to support multiple services types within a server process. Each server type is represented as a class in the server process. Message headers for services requested by the client to the server process include information describing the type of service requested, the specific service requested, and the group Id of the requester. The Object

---

[24] The random number generator must support a wide variety of statistical distributions.

[25] Various types of string classes are provided including variable-length strings, fixed-length strings, and XDR strings.

[26] A good example of a reusable motion library is the Common Algorithm Software Services (CASS) library in JSIMS. It supports a wide range of motion types in four different coordinate systems (ECR, ECI, Round Earth, and WGS84).

[27] High performance scalable communications across wide area networks is an extremely active area of research and development within the network community. It is anticipated that the Network Communications layer will benefit from this ongoing work. By standardizing the distributed network interfaces, and by describing how they are used in the Standard Simulation Architecture, the networking community will be more focused in their research efforts.

[28] Radiant Mercury can be integrated with routing servers to provide multilevel security between networks at different security levels.

Request Broker (ORB) in the server process automates method invocations for server objects. Applications never actually deal with low-level messages.

Table 2 shows performance measurements for the RAM ORB distributed client/server framework involving different configurations. In these measurements, up to 75,000 short messages can be exchanged between two machines on a gigabit Ethernet. Sustained bandwidth for larger messages was measured to be about 15 megabytes per second.[29]

Table 2: Network communication performance between two processors on the same machine, and between two machines.

Network Performance

Configuration	Test Name	Description	Performance
Local Client-Server Dual 1.8 Ghz Linux PC	Synchronous Ping Pong	Client sends ping message and waits for server to respond with pong. Process repeats for 5 seconds.	38,000 messages per second
	Synchronous Variable-length Data	Client sends ping message with variable-length data and waits for server to respond with pong. Process repeats for 5 seconds.	35,000 messages per second
	Asynchronous Ping Pong	Client repeatidly sends ping messages for 5 seconds. Server responds to Pings by sending Pong messages back to the client. The client consumes Pong messages as they arrive until all Pongs are received.	71,000 messages per second
	Bandwidth	Client synronously sends 1 megabye Ping message to the server and waits for the 1 megabyte Pong reply from the Server.	111 megabytes per second

Configuration	Test Name	Description	Performance
Client 1.8 Ghz Linux PC Server 2 Ghz Linux PC Network 1 gigabit Ethernet	Synchronous Ping Pong	Client sends ping message and waits for server to respond with pong. Process repeats for 5 seconds.	14,000 messages per second
	Synchronous Variable-length Data	Client sends ping message with variable-length data and waits for server to respond with pong. Process repeats for 5 seconds.	13,000 messages per second
	Asynchronous Ping Pong	Client repeatidly sends ping messages for 5 seconds. Server responds to Pings by sending Pong messages back to the client. The client consumes Pong messages as they arrive until all Pongs are received.	75,000 messages per second
	Bandwidth	Client synronously sends 1 megabye Ping message to the server and waits for the 1 megabyte Pong reply from the Server.	15 megabytes per second

6.2.5 Internal High-Speed Communications

The Internal High-Speed Communications layer defines the standard set of interfaces that are required to provide high-speed message passing through shared memory and/or through high-speed networks. Multi-node Federates communicate internally through this layer. The Internal High-Speed Communications layer may use services provided by the Network Communications layer to join multiple parallel machines in a network environment. The basic categories of service are described below.

1. Startup and terminate functions to fork processes, create internal shared memory segments, etc., and then to clean up shared memory segments when the Federate exits.
2. Node information to provide the number of nodes (e.g., a UNIX process) and the node Id (ranging from zero to the number of nodes minus one).
3. Synchronization operations to support blocking synchronizations and split-phase fuzzy barrier synchronizations that allow processing to continue while waiting for synchronizations to complete.
4. Global reductions to support basic operations for determining the minimum, maximum, and sum of integer or floating point values provided by each node. A general reduction service must also be provided to support applications that perform general reductions on arbitrary data types.
5. Synchronous data distribution services for broadcast, scatter, gather, and vector/matrix formation.
6. Asynchronous message passing services between nodes. Unicast, destination-based multicast, and broadcast capabilities must be provided.
7. Coordinated message passing services between nodes that guarantee the receipt of all messages before completing the coordinated message-passing operation. Unicast, destination-based multicast, and broadcast capabilities must be provided.
8. Remote method invocation services between objects residing on different processors. Unicast, multicast, or broadcast messaging services must be provided.

Performance benchmarks have been collected using the WarpIV High Speed Communications library (see Figure 10 and Figure 11).[30] All of the shared memory benchmarks to date show nearly perfect

[29] Network performance is highly sensitive to the speed of the network card used by each computer.

[30] These measurements were obtained from an older 48-processor HP Superdome computer running about 1/4 the speed of a 500 MHz Pentium 3. Today's machines are expected to perform about 20 times faster.

scalability as a function of the number of nodes. This kind of scalable performance is impossible to achieve on networked systems (e.g., Ethernet) using standard Internet protocols. Furthermore, all messages are transported reliably though the shared memory, which is critical in supporting high-performance computing for parallel federates executing in logical time.

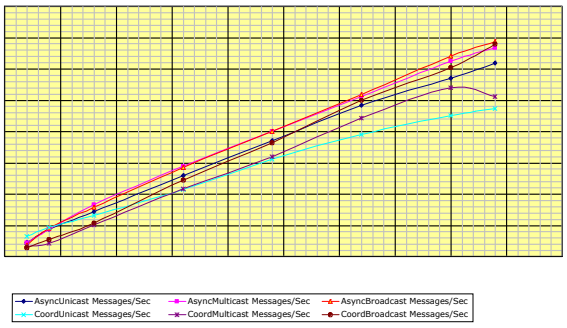


Figure 10: Message throughput performance of the WarpIV High Speed Communications library for message bandwidth on up to 44 processors using shared memory. The throughput scalability is nearly linear as the number of nodes increases.

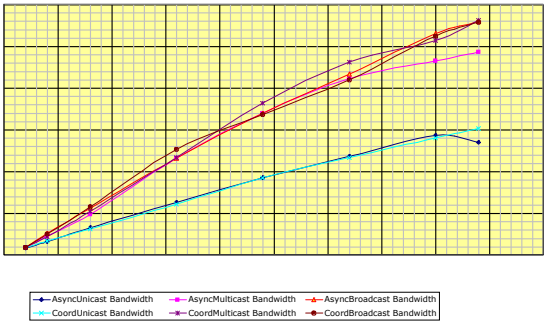


Figure 11: Message bandwidth performance of the WarpIV High Speed Communications library for message bandwidth on up to 44 processors using shared memory. The bandwidth efficiency scalability is nearly linear as the number of nodes increases.

6.2.6 External Distributed Communications

The External Distributed Communications layer defines the standard set of two-way interfaces for communicating between distributed Standard Simulation Architecture Federates or with an external system. The client/server interfaces provided by the Network Communications layer are used to support the various kinds of external distributed communication services required. Note that while these interfaces are defined in this layer, much of their implementation is actually in the Event Management and External Modeling Framework layers.

Services provided by the External Distributed Communications layer include the various command-line utilities that can cause a Standard Simulation Architecture Federation to pause, resume, checkpoint, or exit.

6.2.7 Rollback Framework

The Rollback Framework layer defines the basic interfaces for supporting rollbackable operations. This includes a rollback manager that automatically stores rollback items generated when rollbackable operations are performed. Each event provides its own rollback manager. This allows events to be individually rolled back when necessary.

The framework must efficiently support all rollbackable operations while consuming minimal additional memory. Like standard *undo* and *redo* features provided by most commercial business products, all rollbackable operations must be able to rollback and rollforward without constraints. To minimize processing overheads, rollback items are managed in highly optimized free lists to reduce memory allocation and deallocation times. Each rollback item stores only the information necessary to undo or redo its specific operation.

The rollback framework must also be extensible to allow users to define their own rollbackable operations when necessary. A standard interface has been developed to simplify this task. Users must define a rollback item to undo or redo the operation when necessary. The rollback item inherits from a base class rollback item. Virtual functions are defined on this base class to rollback, rollforward, commit, or uncommit the operation. A simple macro is used to define the memory allocation and deallocation functions. These functions encapsulate the free list

memory management that is critical to maintain high performance execution.

To assist in troubleshooting, the rollback framework also provides diagnostics such as the ability to display all rollbackable operations performed during an event. This information can be printed to the screen, or it can be included in generated trace files that provide information about each processed event.

6.2.8 Rollback Utilities

The Rollback Utilities layer provides a comprehensive set of primitive rollbackable data types, container classes, static and dynamic arrays, standard C functions, I/O services including sorted data logging when executing in parallel, external message passing interfaces, and dynamic memory allocation/deallocation operations.[31] Rollback support can be disabled at run-time through a configuration file setting, or rollbacks can be more optimally disabled through a compile-time flag that eliminates overheads when executing conservatively or when using software designed for reuse in other applications.

6.2.9 Persistence

Persistence fundamentally keeps track of memory allocations and pointer references within a high-speed internal database linked with applications. Through persistence, an object, and the collection of objects it recursively references, can be automatically packed into a buffer that is written to disk or sent as a message to other computers. Later, that buffer can be used to reconstruct the object and all of its recursively referenced objects. These reconstructed objects are normally instantiated in different memory locations. The persistence framework automatically updates all affected pointer references to account for the new memory locations.

Because of the large number of pointers involved, special optimizations are required for persistent container classes. The individual pointers required for managing persistent data structures such as lists and trees are not actually registered with the database. Instead, the container itself is registered. It has been observed that more than a factor of two in memory reduction and performance improvement is achieved by this optimization.

Persistence must be fully integrated with the rollback framework to automate support for optimistic event processing.[32] Persistence also enables dynamic load balancing algorithms to migrate complex objects to different processors.

Persistence provided by CCSE has been used to successfully support the checkpoint/restart requirements for the JSIMS program.

6.2.10 Standard Template Library

The Standard Template Library (STL) provides a suite of generic container classes and algorithms that have evolved into standard C++ utilities. The Standard Simulation Architecture must provide a fully functional STL that also supports persistence and rollbacks.

The STL data structures that are currently supported by the JSIMS Common Component Simulation Engine (CCSE) include: list, map, multimap, set, multiset, vector, string, queue, stack, and priority queue.[33] All iterator types are supported. Four versions of each container are provided: standard, rollbackable, persistent, and rollbackable-persistent.

6.2.11 Event Management Services

The Event Management Services layer provides the core set of services used by the SSA simulation engine. It provides the internal mechanisms for coordinating startup procedures, event processing, and termination procedures. This layer includes interfaces for defining logical processes, creating and deleting event objects,

[31] Rollbackable dynamic memory allocation and deallocation functions with rollbackable pointers must be integrated with persistence to automate checkpoint restart and object migration functionality.

[32] For example, memory allocation/deallocation operations, pointers, and container classes may be both persistent and rollbackable.

[33] The double-ended queue (deque) data structure was not implemented because it was not used by any of the JSIMS applications. This data structure will be incorporated at a later date.

managing pending and uncommitted event queues, supporting event-message passing, handling event retraction, providing event rollback, and invoking event-processing methods. It also provides support for basic trace file generation and the gathering of internal run-time statistics.

Fault-tolerance is supported through checkpoint/restart capabilities. Checkpoint/restart can also be used to support the multi-replication framework that is currently being developed using WarpIV technology for the Air Force.

The multi-replication framework coordinates with the Event Management Services layer to allow large numbers of replicated simulations to execute within a grid-computing system. Replications may be required to explore parameter spaces, generate Monte Carlo statistical results, or to explore multiple courses of action. To further support real-time decision aid capabilities, the multi-replication framework evolves each replication coherently in time.[34] As live data is received, replications are potentially pruned for those replications whose predictions are now out of sync with the real world. New replications may be launched to ensure that the starting state of each replicated simulation matches the current state of the real world. Otherwise, predicted results will not be useful. To explore what-if excursions, the multi-replication framework can execute different plans at critical future decision points in time, and then later prune those plans that do not meet the objectives.

6.2.12 Time Management

The Time Management layer defines the standard interfaces that are required to support various time management algorithms including those that run sequentially on a single processor, conservatively,[35] optimistically,[36] or in real time.[37] The time management layer must also provide generic mechanisms to coordinate the advancement of logical or real time with external systems.[38]

SPEEDES, CCSE, and WarpIV all provide sequential, conservative, and optimistic time management capabilities with built-in flow control. The WarpIV simulation kernel uses several new time management algorithms and internal data structures to further lower the overheads associated with sequential, parallel conservative, and parallel optimistic event processing. WarpIV uses adaptive flow control techniques to further limit rollbacks and message retraction to stabilize the performance of poorly balanced simulations.

6.2.13 Standard Modeling Framework

The Standard Modeling Framework (SMF) layer defines the interfaces for scheduling events locally for objects within an entity or for entities potentially residing on different processors. It also defines process model constructs that are used to support interruptible re-entrant events and method invocations on distributed objects. This layer further defines hierarchical "plug-and-play" components with polymorphic methods to facilitate interoperability between sub-models while minimizing software dependencies.

The SMF provides a sensitivity list mechanism that automatically invokes registered methods when specified attributes are modified. This capability is extended to processes to allow them to *wake up* from WAIT statements when arbitrary complex expressions involving attributes are

[34] For example, suppose each replication starts at time 0 and ends at time 1000. Each replication may be executed up to 10 times with time windows [0,100], [100,200], … [900,1000]. A replication checkpoints its state at the end of each execution, which allows it to be restarted later for resumed execution. This "breadth-first" approach for evolving time, rather than "depth-first", saves wasted processing time when replications are pruned. It also provides near-term projections more quickly to the user.

[35] Conservative time managements may impose topology and/or lookahead restrictions.

[36] Optimistic time management may provide message-sending flow control mechanisms to promote rollback stability.

[37] Real time event scheduling uses the wall clock to assign time tags to events. Real-time events may be scheduled for the current wall time or for real-time values in the future.

[38] The HLA gateway is an example of where generic time synchronization services are used. The generic time synchronization services keep the Federate from advancing (i.e., committing events) beyond the granted time provided by the RTI.

satisfied. Because the invoked functions themselves are allowed to modify other variables in sensitivity lists, a powerful capability is provided to support cognitive algorithms, neural networks, and rule-based logic in expert systems. These services are critical for developing reusable Human Behavior Representation (HBR) components.

The SMF provides automatic entity distribution to one or more processing nodes that are defined within a cluster. Scatter, Block, and direct entity placement decomposition algorithms are supported. Object clustering algorithms allow entities to be constructed on specific machines or nodes within a machine to reduce communication overheads. The SMF must support dynamic entity creation and deletion services. Through object-oriented persistence, the SMF must eventually provide the capability to migrate entities from one node to another node. The entity's complex state and its pending events must be packaged into a message that is used to initialize the entity when it is reconstructed. Object migration is coordinated by dynamic load balancing algorithms that plug into the SMF.

The SMF layer supports object composition capabilities to allow users to hierarchically define entities and their components in a flat file or database. The modeling framework automatically distributes, constructs, and initializes the entities. This capability allows entities to be hierarchically constructed from component repositories. Already composed entities can also be saved in entity repositories for later reuse. Construction of these repositories is critical for reducing the cost of developing highly performing simulations.

6.2.14 Distributed Simulation Management Services

The Distributed Simulation Management Services (DSMS) layer mirrors HLA functionality with automated easy-to-use interfaces. It provides a standard set of interfaces for supporting Federation Objects (FOs), Interactions, interest management, and ownership management.

It is critical for the DSMS layer to provide efficient and scalable interest management for FOs and Interactions. Without efficient interest management, performance breaks down quickly when the numbers of entities gets large. It is also critical for interest management algorithms to support multiple resolutions. The interest management computations should be distributed to avoid bottlenecks when executing in a multiprocessing environment. Efficient multicast techniques to distribute the filtered data through shared memory and networks are also critical to reduce message-passing overheads in large systems.

6.2.15 SOM/FOM Translation

The SOM/FOM Translation layer allows Standard Simulation Architecture Federates to work internally with their own SOM while still being able to interoperate with other Standard Simulation Architecture Federates or HLA Federations. If all Federates use the same object model, then this layer can be bypassed. Translations might include the following.

1. Class name translations for FOs and Interactions.
2. Name translations for FO attributes and Interaction parameters.
3. Unit conversions for attributes and parameters.
4. General translations (e.g., {X, Y, Z} → {Lat, Lon, Alt})
5. Split/merge attributes between multiple FOs.[39]
6. Dynamic values (e.g., motion) with predictive contracts[40] that are computed as a function of time.

[39] For example the FOM might provide two attributes within a single FO. The Federate's SOM, however, might choose to provide each of the attributes in separate FOs. The split/merge functionality provides re-mapping of FO attributes to different FOs.

The SOM/FOM translation services provide an API that can be used to define the translations. A translation description file can be used to specify basic translations. However, to support highly complex translations for applications having a SOM that is very different from the FOM, it is critical to preserve the more general programming interface.

Multiple translations may be applied in series to provide step-wise operations. For example, reflected attributes for an object may first be renamed, then translated to the correct units, and then split into two FOs. These ordered operations should be specified separately.

6.2.16 External Modeling Framework

The External Modeling Framework (EMF) provides interfaces to directly connect Federates with external systems such as graphical user interfaces,[41] analysis tools, remote models, and hardware systems (see Figure 12). It does this in a manner that preserves the same basic set of interfaces provided by the Standard Modeling Framework and the Distributed Simulation Management Services layer.

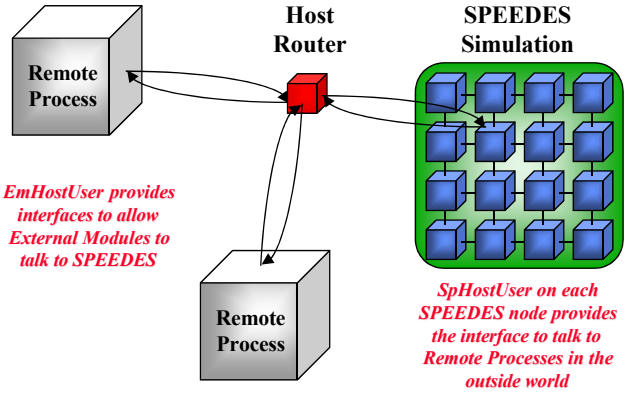


Figure 12: The SPEEDES approach for supporting external systems connecting to a parallel SPEEDES simulation coordinates activities through the Host Router network server. The EmHostUser interface provides the basic interfaces for the remote process, while the SpHostUser provides the interfaces for applications executing within SPEEDES.

External systems using the EMF can schedule events for themselves, and they can schedule or receive events from the simulation. Both logical-time and/or real-time synchronization is provided to ensure that the external module maintains consistency with the simulation. Multiple external systems can connect to an SSA federation. Fault tolerance in the time management synchronization allows the external system to dynamically connect or disconnect without disrupting the overall flow of time in the integrated simulation.

The EMF provides rollbackable state management that can be directed to go forward or backward in time. This is critical for scientific data visualization, real-time analysis, and after-action-review capabilities. With received message capturing, the EMF provides a useful capability to facilitate offline analysis or After Action Review (AAR). Data mining algorithms can be used with the EMF to search for unexpected patterns in a simulation exercise.

6.2.17 Component Repository

The Component Repository layer, populated by model developers, provides a set of reusable components with standardized polymorphic interfaces that can be used to compose entities with models of different or mixed resolutions. Examples of such components might include models of motion, human behavior, sensors,

[40] Predictive contracts publish time-based equations as attributes to reduce update rates. The computed value received by subscribers must be within a specified tolerance as defined by the agreed upon *predictive contract*. The equations may change several times between updates, but the computed values performed by subscribers must remain within the specified tolerance.

[41] A good example of a graphical interface using the External Modeling Framework on JSIMS is the Model Drive Diagnostic Interface (MDDI) tool that allows users to view the attributes of Federation Objects and the parameters of Interactions.

trackers, weapon systems, network communications, guidance systems, logistics, environment, command and control, etc.

The development of component repositories is one of the critical goals of the SSA. This allows entities to be composed by non-programmers when constructing new simulations for supporting studies. Analysts simply decide which components to use when specifying an entity. Different resolution models can be mixed or matched depending on the goals of the simulation exercise.

Keep in mind that the overhead associated with supporting interoperating components is up to six orders of magnitude lower than the overhead between interoperating federates. Components, and their use of polymorphic methods, allow a user to quickly construct a high-performance simulation in a way that was never before achievable through HLA alone.

6.2.18 Entity Repository

The Entity Repository layer provides a library of reusable entities that can be easily instantiated in different simulation applications.[42] Entities may include ships, aircraft, tanks, ground units, bridges, command and control centers, etc.

Entities are defined in the repository by their hierarchical component structure and by their initial parameter settings. The entity repository reuses the effort in defining entities. For example, an F-15 model might be constructed once and then reused in many different simulation applications. There may eventually be several F-15 entity models, each uniquely composed of components to obtain different performance or behavior characteristics. The various F-15 models will likely share many of the same components, but each might have different weapon systems, sensor systems, flight dynamics, or human behavior representations.

[42] A good example of entity models that naturally fit in the Entity Repository is the representation of the environment. Objects such as bridges, roads, weather, and even the terrain are best represented as special types of entities that can publish their attributes to subscribing entities through FOs.

6.2.19 CASE Tools

The CASE Tools layer provides a graphical interface to the Standard Simulation Architecture in order to facilitate higher-level representations of models through code generation and object composition. Code generation has the benefit of reducing human software development errors by automating routine tasks that are error-prone.[43]

The CASE Tools layer may provide API compatibility layers that map interfaces from legacy simulations to the Standard Simulation Architecture.[44] Specialized simulation languages such as Verilog and VHDL may also be provided in this layer.

The CASE Tools layer may include graphical tools to support scenario generation and entity/component model compositions with mixed levels of resolution. It may provide a graphical interface to the component and entity repositories. CASE Tools can also assist in coordinating usage of the SOM/FOM Translation services.

One of the important benefits of standardizing the simulation architecture is that CASE tools can be built by industry with the understanding that their tools will have a market beyond any specific program. This again lowers the cost of developing, composing, executing, and analyzing the generated results of simulations.

6.2.20 HPC-RTI Interface

This layer provides a direct HLA interface to the Standard Simulation Architecture in order to facilitate interoperability with legacy simulations that have their own simulation engine. The benefit provided by this layer is to reduce communication overheads by using shared memory, while additionally providing time managed HLA services for Declaration Management, Data Distribution Management, and Ownership

[43] For example, code generation can ensure that all internal entity state variables are rollbackable when executing optimistically.

[44] The JSIMS Compatibility Layer (JCL) was developed for WARSIM to migrate legacy models to the Common Component Simulation Engine. A prototype JMASS Compatibility Layer was developed for SPEEDES in 2001. A CCSE-SPEEDES Compatibility Layer should be provided to provide backward compatibility for DoD models developed in JSIMS.

Management. Real-time Federates automatically use the wall clock to assign logical time tags to events.

The HPC-RTI can be used to speed up simulations on high performance computing platforms. Its support of time management across all services makes it an ideal choice for analytical simulations that require more processing horsepower to speed up executions.

6.2.21 HLA Gateway

The HLA Gateway provides connectivity to existing HLA Federations using any standard HLA-compliant RTI. It coordinates the flow of data (i.e., Federation Objects and Interactions) through the RTI while also coordinating the advancement of logical and/or real time.

The gateway is implemented as a simulated entity that publishes and subscribes Federation Objects and Interactions with both the RTI and the DSMS layer. For example, an Interaction received by the gateway from the RTI is passed to entities within the Federate by scheduling the Interaction in the DSMS. Similarly, the gateway forwards Interactions through the RTI as it receives them from the DSMS. The gateway may also provide multilevel-security services such as data encryption and security markings for Interactions.

7 Standardization Strategy

The standardization strategy requires the formation of several working groups. First, a *Joint Government Sponsor* should be established to fund and oversee the development of the standard. The Joint Government Sponsor has final decision-making authority over any disputes that may arise during the design and review process. The Government Sponsor may also provide support for basic software processes, configuration management, and formal documentation.[45]

Second, a small *Engineering Team* comprised of proven simulation technologists should be formed to define the initial standard for each of the layers in the architecture. This engineering team should consist of recognized experts from the commercial sector, government laboratories, and academic institutions.[46] Prototype software implementing the layers should be developed to ensure that the initial standard is consistent. The standard interfaces for each layer should be jointly designed by the three groups and then independently developed by each group to validate the standard.

Both *unit* and *system* test suites should be jointly developed by all three groups to ensure that the standard interfaces are implemented correctly. The layers should be standardized starting from the bottom, working upwards until all of the layers are designed, developed, and validated with multiple implementations constructed from within the engineering team. A spiral engineering process is used to permit the refinement of lower layers as the upper layers are developed.

An independent *Technology Panel* of experts[47] will review the standard prototype to ensure that the prototype architecture is robust. Iterations on the standard may occur as recommendations are suggested.

A *User Group*[48] generates feedback on the services provided by the standard. The goal is to attain *buy-in* from the User Group. Again, iterations on the standard may occur as recommendations are suggested.

[45] Software engineering processes should support the Capability Maturity Model (CMM) key process areas through level three.

[46] The commercial sector provides COTS proprietary software products that are licensed to users through purchases, maintenance contracts, etc. The government laboratories provide GOTS software that is free for use on government projects only. For GOTS software to succeed, a government sponsor must provide life-cycle support. Academic institutions may develop open-source software that is generally maintained by grass-roots user communities. All three software business models are represented by this strategy.

[47] Members of the Technology Panel are specialists in specific layers of the Standard Simulation Architecture. Unlike the engineering team, these experts may not have a full background in all of the layers.

[48] Members of the User Group should represent projects that have a vested interest in using the Standard Simulation Architecture.

Once the prototype architecture becomes stable (i.e., no further changes are recommended by the Technology Panel or the User Group), the architecture should go through the formal *IEEE Standardization Process*. At this point, all parties[49] interested in the standard are invited to participate in the further standardization of the layered Standard Simulation Architecture.

8 Benefits

A number of important benefits will be provided to the DoD simulation community once the standardization process is completed.

1. A common infrastructure will exist to facilitate the development of reusable Federates, Entities, and Components.
2. The layered simulation architecture will allow simulation projects to individually combine the most efficient implementations of each layer on targeted machines to achieve the best performance.
3. Optimized sequential and parallel processing capabilities will provide efficient usage of CPU resources ranging from single processor desktop machines to massively parallel supercomputers.
4. A cost effective strategy is provided to focus applied research and development efforts.
5. High-speed interoperability between new models and legacy systems will be fully supported.
6. Software models will be portable to different machines, operating systems, networks, languages, and compilers.
7. The popular business models (i.e., COTS, GOTS, and Open Source) for software development are not only supported, but also encouraged.

[49] New parties interested in participating in the IEEE standardization process may include the entertainment industry, hardware vendors, and Operating System vendors.

Through the formation of standards, the Standard Simulation Architecture will significantly lower the cost of developing, composing, and executing simulations. It will also focus both technology and model development software for reuse, providing synergy in the DoD simulation community.

9 Summary

This paper first provided an overview of the critical issues relating to interoperability and reuse, showing how interoperability concepts from HLA can also be applied to entities and components. High performance is achieved by recognizing the different levels of granularity between interacting federates (milliseconds), entities (microseconds), and components (nanoseconds).

A high-level composable modeling methodology was introduced, showing the relationship between HLA federations, HLA federates, SSA Federations (which are a special kind of HLA federate), HPC-RTI federates (which operate inside the SSA), SSA Federates, entities, components, and federation objects. Another view describing system composability focused on inter-processor communications considerations involving threads, nodes, machines, shared memory, local area networks, and wide area networks.

This paper then provided an outline of the Standard Simulation Architecture that is based on experience from several large DoD simulation programs. The Common Component Simulation Engine (CCSE) developed on the Joint Simulation System (JSIMS) provides the starting point for the requirements and initial implementation of the standard.

A dependency-layered approach is used to describe the software libraries that comprise the architecture. This allows technology providers (i.e., universities, government laboratories, and commercial organizations) to develop optimized libraries for the overall system. These libraries can plug and play together because the interfaces

are standardized. The layered approach will help focus and apply R&D efforts for transition onto real DoD simulation programs.

The proposed Standard Simulation Architecture promotes COTS, GOTS, and Open Source business models, thereby providing a cost-effective way for commercial, government, and academic institutions to participate in developing common models, tools, and simulation technology for reuse on multiple DoD simulation programs.

An outline for the standardization strategy was provided. A joint government sponsor is needed to oversee and manage the overall process. Three teams are required to develop the standard. The Engineering Team should be comprised of members from industry, academia, and government laboratories to develop and test prototypes of the standard layers. The Technology Panel should be comprised of experts and specialists in critical areas of the architecture. Their job is to ensure that the right technologies are applied to the architecture. The User Group is comprised of actual model developers and simulation users. Their job is to generate requirements and approve of the capabilities being developed.

10 Acknowledgements

This work was sponsored by the SPAWAR System Center in San Diego (SSC-SD) through its development of the SPEEDES-based Common Component Simulation Engine (CCSE) in support of the Joint Simulation System (JSIMS). This work was also funded in part by the High-Performance Computing Modernization Program (HPCMP) through its Common HPC Software Support Initiative (CHSSI).

11 Authors

Jeffrey S. Steinman

Dr. Jeffrey S. Steinman, Vice President and Chief Scientist at RAM Laboratories received his Ph.D. in 1988 from the University of California Los Angeles in High-Energy Particle Physics. Between 1988 and 1995, Dr. Steinman led several high-performance computing R&D activities at JPL/Caltech in support of Strategic Defense, Air Defense, Ballistic Missile Defense, and NASA space exploration missions.

While at JPL/Caltech, Dr. Steinman pioneered the technology and software development of the Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) framework. This work resulted in five patents and more than forty technical papers in the area of high-performance computing, optimistic discrete-event simulation, data structures, message-passing algorithms, object-oriented design, parallel and distributed multi-resolution interest management algorithms, and HLA.

Dr. Steinman directed software engineering teams that developed core infrastructures for several mainstream simulation projects including the Parallel and Distributed Computing Simulation (PDCS), the Parallel Naval Simulation System (NSS), Wargame 2000 (WG2K), the Joint Simulation System (JSIMS), the Joint Modeling and Simulation System (JMASS), High-Performance Computing Run Time Infrastructure (HPC-RTI), and the Human Behavior Representation Testbed (HBR-Testbed). Dr. Steinman was a member of the HLA Time Management and Data Distribution Management working groups and wrote the original design document for DDM. Dr. Steinman is currently the lead architect for the JSIMS Common Component Simulation Engine.

Douglas R. Hardy

Mr. Hardy, Project Manager at Space and Naval Warfare Systems Center, San Diego received his Master's in 1985 from Arizona State University in Applied Mathematics and Physics. From 1986 to the present, Mr. Hardy, has led several Modeling and Simulation R&D projects in support of the Defense Advanced Research Projects Agency (DARPA), the Office of Naval Research (ONR), Naval Health Research Center (NHRC), Joint Simulation Alliance Executive Office (AEO), and a variety of multi-sponsored

Advanced Concepts Technology Demonstrations (ACTDs).

In the early '90's, Mr. Hardy led an effort to link the Army's Constructive Battalion/Brigade Simulation (BBS) with the Army's Virtual Tank Simulators using Distributed Interactive Simulation (DIS) protocols. The primary technical challenge was to allow realistic interoperation between the two disparate systems to support BBS battle staff training with simultaneous tank crew training. The project became the cornerstone of the Synthetic Theater of War – Europe (STOW-E) in November of 1994, which integrated several simulation and instrumented sites in Germany and around the globe.

Since that time, Mr. Hardy has been primarily involved in leading software development projects related to Modeling and Simulation R&D efforts. This has included leading the development of the Navy Semi-Automated Forces (SAF) for the STOW ACTD, leading the development of a mine component in Navy SAF for the Joint Countermine ACTD, supporting the transition of Navy SAF software to Joint SAF (JSAF), leading the development of a medical component of JSAF (called JMedSAF) for the Joint Medical ACTD, leading a Capability Maturity Model (CMM) Level 3 development effort for the Common Component Simulation Engine (CCSE) for the Joint Simulation System (JSIMS) program, and currently leading a modernization development effort for the Enhanced Naval Warfare Gaming System (ENWGS) program.

12 BIBLIOGRAPHY

1. Bailey Chris, McGraw Robert, Steinman Jeff, and Wong Jennifer, 2001. "SPEEDES: A Brief Overview" In proceedings of *SPIE, Enabling Technologies for Simulation Science V*, Pages 190-201.
2. Cassandras Christos, 1993. "Discrete Event Systems, Modeling and Performance Analysis." Aksen Associates Incorporated Publishers, IRWIN, Homewood, Illinois 60430.
3. Chandy, K., and Misra, J., 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." IEEE Transactions on Software Engineering. Vol. SE-5, No. 5, pages 440–452.
4. Clark Joe, Capella Sebastian, Bailey Chris, Steinman Jeff and Peterson Larry, 2002. "The Development of an HLA Compliant High Performance Computing Run-time Infrastructure" In proceedings of the *2002 Spring Simulation Interoperability Workshop, Paper 02S-SIW-016*.
5. Deitel & Deitel, 2001. "C++ How to Program, Third Edition." Prentice Hall, Inc., Upper Saddle River, New Jersey 07458.
6. Federal Information Processing Standards, 1994. "Guideline for the Analysis Local Area Network Security." Publication 191, http://csrc.nist.gov/publications/fips.
7. Fogel Karl, 1999. "Open Source Development with CVS." The Coriolis Group, LLC, 1445 North Hayden Road, Suite 220, Scottsdale, Arizona 85260.
8. Fowler Martin and Kendall Scott, 1999. "UML Distilled, Second Edition, A Brief Guide to the Standard Object Model Language." Addison-Wesley Longman, Inc.
9. Fujimoto Richard, 2000. "Parallel and Distributed Simulation Systems." John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012.
10. Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, 1995. "Design Patterns, Elements of Reusable Object Oriented Software." Reading, MA, Addison-Wesley Publishing Company.
11. Gilb Tom and Graham Dorothy, 1998. "Software Inspection." Addison-Wesley Longman Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England.
12. Gregory Peter, 1999. "Solaris Security." Prentice-Hall PTR.

13. Goldfarb Charles and Prescod Paul, 1998. "The XML Handbook." Prentice-Hall PTR, Upper Saddle River, NJ.

14. Harold Elliotte, 1999. "XML Bible." IDG Books Worldwide, 919 E. Hillsdale Blvd., Suite 400, Foster City, CA 94404.

15. http://standards.ieee.org/resources/glance_at_nescom.html, "The Project Authorization Request."

16. ISO/IEC, 1998. "Programming Languages – C++." American National Standards Institute, 11 West 42nd Street, New York, NY 10036.

17. Jefferson David, 1985. "Virtual Time." ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pages 404-425.

18. Joint Simulation System (JSIMS) System Subsystem Design Document.

19. Joint Simulation System (JSIMS) Common Component Simulation Engine (CCSE) Software Design Document (SDD).

20. Joint Simulation System (JSIMS) Common Component Simulation Engine (CCSE) Software User Manual (SUM).

21. Joint Simulation System (JSIMS) Common Component Simulation Engine (CCSE) Interface Requirement Specification (SDD).

22. Kanarick C. 1991. "A Technical Overview and History of the SIMNET Project." In Proceedings of the 1991 Advances in Parallel And Distributed Simulation Conference, Pages 104-111.

23. Kuhl Frederick, Weatherly Richard, and Dahmann Judith, 2000. "Creating Computer Simulation Systems, An Introduction to the High Level Architecture." Prentice Hall PTR, Upper Saddle River, NJ 07458.

24. Lee James, 1999. "Verilog Quickstart! A Practical Guide to Simulation and Synthesis in Verilog." Kluwer Academic Publishers, 1001 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061.

25. Lewis Bil and Berg Daniel, 1996. "Threads Primer, a Guide to Multithreaded Programming." Prentice-Hall PTR, One Lake Street, Upper Saddle River, NJ 07458.

26. McGuinness Terry, Bailey Chris, Landa Chris, Steinman Jeff, Peterson Larry, 2002. "Executing Independent Parallel Applications Using the SPEEDES Communications Library." In proceedings of the *High Performance Computing Modernization Program Users Group Conference*.

27. Meyer Bertrand, 1997. "Object Oriented Software Construction, Second Edition." Prentice Hall Professional Technical Reference.

28. Mikkelsen Tim and Pherigo Suzzanne, 1997. "Practical Software Configuration Management." Hewlett-Packard Professional Books, Prentice-Hall PTR, Upper Saddle River, NJ 07458.

29. Misener Steven and Krawetz Stephen, 2000. "Bioinformatics Methods and Protocols." Humana Press Inc., 999 Riverview Drive, Suite 208, Totowa, New Jersey 07512.

30. Molloy Michael, 1989. "Fundamentals of Performance Modeling." Macmillan Publishing Company, 866 Third Avenue, New York, New York 10022.

31. Morse Katherine, Steinman Jeff, 1997. "Data Distribution Management in the HLA: Multidimensional Regions and Physically Correct Filtering." Spring *Simulation Interoperability Workshop*, No. 97S-SIW-052.

32. Nilsson Nils, 1993. "Principles of Artificial Intelligence." Morgan Kaufmann Publishers, Inc.

33. Paulk Mark, 2001. "Extreme Programming from a CMM Perspective. "Paper for XP Universe, Raleigh NC, 23-25 July 2001.

34. Paulk Mark, Curtis Bill, Chrissis Mary Beth, and Weber Charles, "Capability Maturity Model for Software, Version 1.1", Software

Engineering Institute, CMU/SEI-93-TR-24, DTIC Number ADA263403, February 1993.

35. Paulk Mark, Weber Charles, Garcia Suzanne, Chrissis Mary Beth, and Bush Marilyn, "Key Practices of the Capability Maturity Model, Version 1.1", Software Engineering Institute, CMU/SEI-93-TR-25, DTIC Number ADA263432, February 1993.

36. Perry Douglas, 1994. "VHDL." McGraw-Hill, Inc., 1221 Avenue of the Americas, New York, NY 10020.

37. Phillips Dwayne, 1998. "The Software Project Manager's Handbook, Principles that Work at Work." IEEE Computer Society Press Customer Service Center, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1314.

38. Plauger P.J., Stepanov Alexander, Lee Meng, Musser David, 2001. "The C++ Standard Template Library." Prentice Hall PTR, Prentice-Hall Inc., Upper Saddle River, NJ 07458.

39. Pullen Mark, Myjak Michael, and Bouwens Christina, 1999. "Limitations of Internet Protocol Suite for Distributed Simulation in the Large Multicast Environment." RFC –2502, http://www.ietf.org/rfc/rfc2502.txt?number=2502.

40. Pyarali Irfan, Schmidt Douglas, and Cytron Ron, 2002. "Achieving End-to-End Predictability of the TAO Real-time CORBA ORB." Submitted to the *8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, September 2002.

41. Raymond Eric, 2000. "The cathedral and the bazaar." http://tuxedo.org/~esr/writings/cathedral-bazaar/hacker-history/.

42. Reilly Sean and Briggs Keith (Editors), 1999. "Guidance, Rationale, and Interoperability Modalities for the Real-Time Platform Reference Federation Object Model (RPR-FOM) Version 1.0." Simulation Interoperability Standards Organization.

43. Reynolds Paul, 1991. "An Efficient Framework for Parallel Simulations." In Proceedings of the *Advances in Parallel and Distributed Simulation Conference* (PADS91). Vol. 23, No. 1, January 1991, Pages 167-174.

44. Rolston David, 1988. "Principles of Artificial Intelligence and Expert Systems Development." McGraw-Hill, Inc.

45. Schildt Herbert, 1998. "C++: The Complete Reference, Third Edition." Osborne McGraw-Hill, 2600 Tenth Street, Berkeley, California 94710.

46. Schmidt Douglas and Kuhns Fred, 2000. "An Overview of the Real-time CORBA Specification." IEEE Computer special issue on Object-Oriented Real-time Distributed Computing, edited by Eltefaat Shokri and Philip Sheu, June 2000.

47. Schneier Bruce, 1995. "Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition." Wiley, John & Sons Incorporated.

48. Schuylmeyer Gordon, Mackenzie Garth, 2000. "Verification & Validation of Modern Software-Intensive Systems." Prentice Hall, Inc., Upper Saddle River, NJ 07458.

49. SPAWAR Systems Center, San Diego, 2000."Software Management for Executives Guidebook." PR-SPTO-03-V1.7, September 1, 2000. Software Engineering Process Office (SEPO) D12.

50. Sperber Joan, 2001. "Up to SPEEDES." Military Training Technology, MT2, Volume 6, Issue 1, 2001.

51. Steinman Jeff, 1993. "Breathing Time Warp." In proceedings of the *7th Workshop on Parallel and Distributed Simulation (PADS93).* Vol. 23, No. 1, Pages 109-118.

52. Steinman Jeff, Nicol David, Wilson Linda, and Lee Craig, 1995. "Global Virtual Time and Distributed Synchronization." In proceedings

of the *1995 Parallel And Distributed Simulation Conference*. Pages 139-148.

53. Steinman Jeff, 1998. “Scalable Distributed Military Simulations Using the SPEEDES Object-Oriented Simulation Framework.” In the proceedings of the *Object-Oriented Simulation Conference* (OOS’98), pages 3-23.

54. Steinman Jeff, Tran Tuan, Burckhardt Jacob, Brutocao Jim, 1999. “Logically Correct Data Distribution Management in SPEEDES”, In proceedings of *the1999 Fall Simulation Interoperability Workshop*, Paper 99F-SIW-067.

55. Treshansky Allyn and McGraw Robert, 2002. "MRMAide[tm]: A Mixed Resolution Modeling Aide." In proceedings of *SPIE, Enabling Technology for Simulation Science VI*.

56. Tung Yu-Wen and Steinman Jeff, 1993. “Interactive Graphics for the Parallel and Distributed Computing Simulation.” In proceedings of the *1993 Summer Computer Simulation Conference*. Pages 695-700.

57. U.S. Department of Defense, “High Level Architecture Interface Specification, Version 1.3.”

58. U.S. Department of Defense, “High Level Architecture Object Model Template, Version 1.3.”

59. U.S. Department of Defense, “High Level Architecture Rules, Version 1.3.”

60. Valinski Maria, Driscoll Jonathan., McGraw Robert, and Buchy Doug, 2001. "Providing JMASS with a Distribution Simulation Capability Using SPEEDES." In proceedings of the *Summer Computer Simulation Conference*.

61. Wallace Jeff, et al., 1999. “IMPORT V2.0 Beta 1: A Tool for Large Scale, Complex System Simulation.” In proceedings of the *High Performance Computing 1999, Grand Challenges in Computer Simulation Conference*, Pages 267-272.

62. Weatherly R., Wilson A., Griffin S., 1993. “ALSP – Theory, Experience, and Future Directions.” In Proceedings of the *1993 Winter Simulation Conf4erence*, Pages 1068-1072.

63. Wieland Fred et al. 1989. "The Performance of a Distributed Combat Simulation With the Time Warp Operating System." *Concurrency: Practice and Experience*. Vol. 1, No. 1, Pages 35-50.

64. Wilson Linda and Nicol David, 1995. "Automated Load Balancing in SPEEDES." In proceedings of the *1995 Winter Simulation Conference*, Pages 590-596.

65. Zeigler Bernard, Praehofer Herbert, Kiim Tag Gon, 2000. “Theory of Modeling and Simulation.” Academic Press, 525 B Street, Suite 1900, San Diego, CA 92101-449