A Proposed Open System Architecture for Modeling and Simulation (OSAMS)

In support of the Joint Program Executive Office for Chemical and Biological Defense (JPEO-CBD)
Software Support Activity (SSA)
Modeling Simulation and Integration (MSI) Team

Jeffrey S. Steinman, Ph.D.
WarpIV Technologies, Inc.
(858) 531-0643
steinman@warpiv.com

Jennifer Park
SPAWAR Systems Center, San Diego (SSC-SD)
(619) 553-2848
jennifer.park@navy.mil

Bruce “Wally” Walter
L3 – Titan
(619) 553-4013
walwal@nosc.mil

Nathan Delane
EG&G Technical Services
(540) 663-9426
ndelane@egginc.com

Keywords:
OSAMS, OpenMSA, HLA, Components, SOA, M&S, VV&A, Interoperability and Reuse, Distributed Objects, SPEDEES, WarpIV

ABSTRACT: This paper describes how modern component-based M&S interoperability technologies can be standardized to significantly lower the development, operation, and life-cycle maintenance costs of next generation models for the Department of Defense (DoD) and industry. While current standards focus on simulation-to-simulation interoperability in network environments, the Open System Architecture for Modeling and Simulation (OSAMS) is primarily focused on standardizing the interfaces used by model developers to promote robust model-level interoperability. OSAMS provides a Service Oriented Architecture (SOA) for Modeling and Simulation. As a proposed standard, OSAMS ought to be independent of any particular simulation engine implementation, and should thereby promote the integration of simulation technologies from private industry, government research laboratories, mainstream defense programs, and academic institutions. OSAMS is designed to automate integration with current simulation-based interoperability standards such as the High Level Architecture (HLA), Distributed Interactive Simulation, and the Test and Training Enabling Architecture (TENA). OSAMS is also designed to support emerging web-based standards such as the Extensible Modeling and Simulation Framework (XMSF), Simulation Reference Markup Language (SRML), Base Object Models (BOM), and Network Centric Enterprise Services (NCES) that will eventually provide connectivity between simulations, databases, and operational systems across the Global Information Grid (GIG). OSAMS is part of the Open Modeling and Simulation Architecture (OpenMSA) that will be investigated as part of the Open Source Initiative for Parallel and Distributed Modeling and Simulation (OSI-PDMS). This has been proposed as a new study group within the Simulation Interoperability Standards Organization (SISO).

1 Need for New Standards

Interoperability and reuse has been a major pursuit in lowering the cost of Modeling and Simulation (M&S) over the last twenty years. Interoperability technologies such as the Aggregate Level Simulation Protocol (ALSP) [56], Distributed Interactive Simulation (DIS) [5,7,14], and the High Level Architecture (HLA) [21,22,23,24,30] have taken a distributed computing approach where individual simulators are combined to form a distributed simulation system. Emerging technologies such as the Extensible Modeling and Simulation Framework (XMSF) [11] and the Simulation Reference Markup Language (SRML) [55] have directly integrated standard web services into their interoperability architectures. The Base Object Model (BOM) [1,20] standard applies HLA, Extensible Markup Language (XML) [49] and Unified Modeling Language (UML) [12] technologies to describe the interfaces and roles of interacting models. Network Centric Enterprise Services (NCES) [5] standards are currently being implemented to support Network Centric...
Operational Warfare (NCOW) [35] on the Global Information Grid (GIG) [4,17,18]. The Test and Training Enabling Architecture (TENA) [47] provides communications infrastructure to integrate live and simulated entities in range testing. Standardized data models [5,37,25] have been established to work with these interoperability frameworks.

In all of these federated or enterprise systems, network-based message-passing services are provided by an agreed-upon communications core infrastructure such as the HLA Run Time Infrastructure (RTI). Sharing a common run time infrastructure allows the individual simulations to define and exchange state data, schedule mutual interactions, and collectively coordinate the advancement of simulated time [13]. Each simulation is responsible for coordinating its internal logical and/or real time event processing with the rest of the federation. To support logical time synchronization, the run time infrastructure asynchronously receives time advance requests from each simulation. Based on these time advance requests, time advance grants are independently provided to the simulations in a manner that ensures causality as each simulation processes its own internal events and as they mutually interact.

The network services do not provide the application-level event scheduling constructs that are necessary to develop actual models. Each simulation is free to choose whatever simulation engine and/or event scheduling strategies are required to support its internal event processing. Middleware solutions are frequently employed to simplify the integration of simulations with interoperability run time infrastructures such as HLA or DIS [42]. In other cases, the simulation engine itself provides automatic interoperability with these standards. Such simulations are usually designed to run in both (1) standalone and (2) federated modes of operation.

The federation approach has had much success, but its use also comes with a very high cost. Interoperability and reuse today is still far too expensive to facilitate widespread use of modeling and simulation for acquisition, training, test & evaluation, and operational planning with faster-than-real-time prediction. Adapting simulations to interoperate with other simulations through HLA requires a significant software development investment [58]. Often, the underlying interoperability infrastructure is inappropriate for the intended use (e.g., using real-time DIS or HLA to support Monte Carlo analysis requiring large numbers of replications).

While simulation interoperability technologies today still offer significant cost reductions over live testing, and in some cases it is the only feasible way to conduct experiments and studies, it is still far too expensive. This paper focuses on how to significantly lower the cost of modeling and simulation through standards that support direct reuse at the model-component level. This is important because ultimately, the high cost of simulation impacts the warfighter’s ability to successfully complete his or her mission.

In addition to high costs, the current federated approach also poses several technical problems that can sometimes severely limit the performance and/or fidelity of the actual exercise. This means that modeling and simulation users must sometimes choose between either: (1) obtaining results in a timely manner, or (2) obtaining accurate results, but not both.

![Figure 1: Interoperability through federating vs. direct model composability.](image-url)
Figure 1 highlights some of the problems in federating independent simulations using a commonly accepted standard such as the High Level Architecture. This is in contrast to the direct model composability and integration techniques recommended in this paper that would directly link interoperable models into a single program operating within a common standardized execution environment.

The Open System Architecture for Modeling and Simulation (OSAMS) described in this paper also supports integration with existing standards such as HLA for legacy simulations, but recommends developing new overarching models with a proposed plug-and-play methodology that facilitates interoperability through standards supporting direct model composability.

While each individual simulation application in practice experiences its own interoperability and integration issues, some of the common challenges of developing distributed federations composed of multiple simulations [3] are briefly described below:

1. Integration costs for federating simulations can be expensive to develop, maintain, and modify. Integrating disjoint simulations with distributed interoperability mechanisms can also severely increase the complexity of the software models themselves, thus raising overall maintenance costs. Software models are frequently littered with run time infrastructure calls to coordinate messages, etc. These systems can quickly become unwieldy, especially when supporting both standalone and federated modes of operation.

2. There are no accepted standards for startup and shutdown procedures, which especially affects federations operating in logical time. This means that simulations are typically modified whenever they participate in new federations with different federate members. In addition, large federations often require a team of expert operators to startup, maintain, and shut down their systems in an exercise. Multiple computers are usually required to host the federation. Users of the simulation capability frequently rely on a team of operators to run the exercise.

3. Agreement on the actual data exchanged between the different simulations can be challenging, especially when simulations participate in multiple federations with unique data models or message-passing data formats. Format conversions between big endian and little endian machines must be agreed upon, or text-based formats must be used. Furthermore, packing and unpacking overheads can be very large, especially when text-based parsing formats such as XML are used to exchange data during operation.

4. Simulations must turn off or “ghost” their internal models when their corresponding functionality is provided by other simulations. Not only does this increase the software complexity of the simulation, but it can also be very difficult to accomplish in many cases. In addition, simulations and their models are frequently modified to support new capabilities in acquisition or test and evaluation studies. Because models are often coupled to other models, even minor modifications can require significant software changes, large efforts in testing, and expensive revalidation of the modified models. Usually, expert software developers who are familiar with the idiosyncrasies of each simulation are required to make these changes. Programmers who are not familiar with the design of the simulation often have a difficult time even finding the right set of files that need to be modified. Making the right modifications is even more challenging. An example of this kind of problem is in reconciling communications models, which most campaign-level tools provide in their standalone capability. Often, it is nearly impossible to reconcile the unique behaviors of different communication models implemented in each simulation, especially when the battlefield entities heavily rely on next generation network strategies.

5. Significant network communication overheads impact the run-time performance and/or modeling fidelity of the federation. Real-time simulations must deal with network-based bandwidth and/or latency delays that can be exorbitant during periods of high volume network traffic. In some cases, messages may even be dropped when higher performing, scalable, but unreliable best-effort message-passing services are used. Logical-time simulations must similarly deal with lookahead requirements that constrain how tightly in simulated time the simulations within the federation may interact. Larger lookahead values will improve execution performance, but at the cost of decreasing fidelity between interacting models. Workarounds in dealing with latency or lookahead constraints that try to predict state changes or model interactions can help, but at the expense of further complicating the models and increasing maintenance costs. Such prediction techniques still run into serious problems when the predictions are incorrect due to some unforeseen interaction. Handling these situations makes the models even more complicated and brittle. In some cases, a reasonable compromise between run-time performance and modeling fidelity is not possible. In these circumstances, the federated approach becomes unworkable.

A better approach to providing high performance interoperability and reuse for modeling and simulation is
to exploit plug-and-play component technologies for constructing hierarchical software models. Such techniques have been successfully used in a wide variety of commercial applications including music production, graphic arts, office productivity suites, web browsers, entertainment, etc. If developed with the right methodology, componentized models could be maintained in a model component repository (i.e., a software library) and then directly linked into simulations by scenario developers and operators as needed. Attempts have been made in the past to support this concept, with mixed success. It is the contention of this paper that the full set of capabilities and advanced technologies provided by OSAMS, along with a rigorous model-component software development methodology are required to allow this concept to succeed.

Hierarchical construction of such model components at run time would allow users to quickly construct their simulations from reusable software components in a flexible manner. To compose simulations, users would access a catalog or repository of models that are published and potentially accessed through discoverable downloadable web services. Abstract, but well-defined model interfaces would allow these model components to cleanly interact with one another without introducing strong coupling in their specific software implementations. An ontology that describes model interfaces and the roles they play with other models is necessary to define which models can be effectively combined. The new Base Object Model (BOM) standard that was recently published by the Simulation Interoperability Standards Organization (SISO) may provide the right methodology for characterizing models in terms of composability with other models. Choosing the right methodology for developing model components is critical for maximizing interoperability reuse.

A rich plug-and-play modeling and simulation architecture shared by model developers is necessary to support this concept. It is virtually impossible to develop highly reusable model components without a standard modeling interface. This paper describes how the M&S community can provide such a standard interface for the development of next generation overarching models while simultaneously maintaining interoperability with legacy modeling and simulation capabilities.

The primary goals of OSAMS are to (1) protect the large investments made in systems that are in use today, while (2) providing a roadmap forward for developing next-generation highly interoperable high-performance overarching models. A secondary goal of OSAMS is to provide a well-structured and focused Science & Technology program for developing next-generation standards-based simulation technology. This means that successful simulation architecture Research & Development conducted by Science & Technology programs will directly benefit future programs.

OSAMS is part of a larger layered architecture, known as the Open Modeling and Simulation Architecture (OpenMSA). The OpenMSA focuses on standardizing many of the technologies of parallel and distributed M&S, while OSAMS focuses on standards and methodologies for supporting model interoperability. Other M&S architectures in addition to the OpenMSA could support OSAMS.

## 2 Proposed Solution

One might consider how current operations might change if the cost of modeling and simulation were significantly reduced. A revolutionary, and not evolutionary, approach that focuses on interoperability and reuse through model composability is the key to accomplishing this goal.

The problem is not that we need to things better... We need to do things differently!

Large savings across the full modeling and simulation usage spectrum are possible by standardizing the architecture for model development. Large savings will span costly support and production areas such as model development, unit/system testing, troubleshooting, maintenance, operational training, Verification Validation and Accreditation (VV&A), test and evaluation, scenario generation, after action review, etc. In addition, through direct plug-in mechanisms, better run time performance may be achieved without sacrificing fidelity.

The proposed architecture will support execution on (1) highly portable computers such as laptops or even PDAs, (2) the world’s most powerful supercomputers, and (3) any other conceivable computing architecture composed of networked machines. This paper discusses the critical plug and play technologies that will enable models to interoperate with other models without requiring strong mutual dependencies. This paper also discusses how to simultaneously maintain interoperability with legacy systems through current standard federation technologies.

Specifically, this paper characterizes a standard modeling and simulation architecture with interfaces that provide all of the necessary constructs to develop highly reusable and interoperable models, while being simple enough to be supported by a variety of modern simulation engines, including several that are freely available today [40,46].

The approach is to define a comprehensive set of interfaces in a newly created Open System Architecture for Modeling and Simulation (OSAMS) that will significantly reduce the effort in model development, testing, validating, and integration. In addition, a software
development methodology and standardized ontology formalism will be provided to guide model developers in building highly interoperable model components.

This paper focuses on five specific areas:

1. **Flexible Hierarchical Composition Structure** with the ability to dynamically define, compose, and construct simulation objects from model components at run time in a distributed environment

2. **Standard Modeling Framework** that allows models to schedule/process events and advance time with the goal of minimizing the amount of code necessary to implement complex models

3. **Abstract Polymorphic Methods** infrastructure and methodology that decouples the dependencies of highly interacting models to promote interoperability and code reuse

4. **Distributed Object Technologies** that allow models to package exportable attributes into federation objects that can be published and/or subscribed in a scalable manner within a distributed environment

5. **Trace File Generation and Data Logging Capabilities** that allow time ordered events in a distributed environment to be recorded and consolidated, along with general-purpose user-provided data that can be logged to support debugging, testing, analyzing, and validating the results of a simulation execution

3 **Technical Background**

A good place to begin when discussing modeling and simulation is to start with some basic definitions.

**Models** include any physical, mathematical, or otherwise logical representation of a system, entity, phenomenon or process.

**Simulations** include a method of implementing a model over time.

DIS Glossary of M&S Terms, DoD Directive 5000.59
DoD Publication 5000.59-P and MSETT NAWC-TSD Glossary

According to this definition, models are essentially static representations that require the notion of simulation to propagate their simulated behavior over time. Unfortunately, this does not describe what model developers actually build. This mismatch can cause problems when engineers try to communicate with program managers and high-level policy makers.

From a software engineer’s perspective, simulations are software programs that are composed of time-evolving models. Correspondingly, models are an abstract software representation of a system. In addition, models may be composed of other models.

A simulation engine\(^1\) usually provides core event scheduling and event processing services that allow models to interact and to perform or coordinate their operations over time (see Figure 2). A good simulation engine dramatically reduces the effort of building highly interoperable models while also providing a much more robust development environment.\(^2\)

![Wall of Separation](image)

**Figure 2:** Wall of separation between the simulation engine and the collection of models it supports.

Skilled modelers very carefully design how their models advance in time to handle complex interactions with other models, minimize event-processing computations, and produce highly accurate results.

**Modeling intrinsically involves coordinating the advancement of simulated time.**

In the simplest time advancement strategy, the simulation engine updates the state of its models in a lockstep loop that increments time and then calls an update method on each model. This simple time-stepped, or for-loop, approach to time management can be useful for some real-time and low-level engineering or physics-based systems. The time-stepped approach also supports the

\(^1\) Some simulations have very mature, sophisticated simulation engines, while other simulations provide homegrown infrastructure that manage the scheduling and processing of events. A poorly designed simulation system blurs the division between the event processing infrastructure and the model software, resulting in costly maintenance, brittle software, and bloated code.

\(^2\) On the Joint Simulation System (JSIMS) program, one of the Development Agencies reported that their original design was estimated to require approximately 25,000 lines of code. Using the advanced features of the JSIMS Common Component Simulation Engine (CCSE), the resulting system was implemented in approximately 4,000 lines of code.
important notion of having passive models that are independent of what techniques are used to advance time. However, a much more powerful and efficient way to implement simulations of complex asynchronously interacting systems is the discrete-event approach.

In discrete-event simulations, events, as they are processed, are free to schedule new events that may occur at any future point in time. Events may never be scheduled in the past because that would violate causality. Because time can take on any value, events in the discrete-event paradigm are allowed to occur at their natural instances in time without being correlated to other independent activities. This leads to better performance and higher model fidelity. However, discrete-event simulations rarely provide standalone black-box models because the models themselves fundamentally coordinate how they evolve over time. This is in stark contrast to the widely understood DoD definition of modeling and simulation where models are thought-of as static and independent of time.

Without a standard architecture such as OSAMS, one might ask what it would take for a software model to truly be reusable...

To begin, a reusable model would have to be self-contained and independent of the simulation engine. It must not actively coordinate how it advances in time in any manner because this would tie it to a particular simulation engine. Even interfaces to obtain the current simulation time would not be allowed. A reusable model would also have to be independent of specific implementations of other models. Hard dependencies between models limit reuse because entire systems of dependent models would have to be reused as a whole. In addition, global variables or other software dependencies must be avoided. All of the state variables within a model would have to be self-contained and encapsulated from other models.

In practice, these interoperability and reuse conditions on models are rarely followed. Most likely, models are tightly coupled to the simulation engine, other models, utility services, and various global variables. To illustrate these interoperability concepts further, consider two very different types of model representations.

1. **Active models**... do things to other models. They could be characterized as verbs. Active models are not reusable because by nature they are strongly coupled to other objects or engine constructs.

2. **Passive models**... are manipulated by active models. They could be characterized as “noun” objects. Passive models are potentially reusable because they can be developed with minimal dependencies on the simulation engine and other models.

The time-stepped approach discussed earlier in this paper updates time in a “for-loop” where time is uniformly advanced for all of the models. An update function updates the state of each model to the new time value. This results in an inherently passive form for models as long as they do not directly manipulate other models. A simple C++ code example, shown in Code Segment 1, indicates how time-stepped simulations advance in time.

**Code Segment 1**: A simple example of a time-stepped simulation is shown. The simulation starts at time 0.0 and updates the state at regular time increments. The simulation ends when it reaches its end time.

```cpp
for (time=0.0; time<endTime; time+=increment) {
    UpdateState(time);
}
```

It is tempting to apply the time-stepped approach in developing models because it is extremely simple to implement. In addition, the time-stepped approach also encourages the development of passive models, which helps support reuse. However, there are severe limitations with this approach. **First**, it forces artificial time granularities into the models. For example, suppose time advances every sixty seconds, but a modeled entity receives weapon fire at arbitrary times. Challenging modeling issues arise in time-stepped simulations when handling asynchronous events that naturally occur in the real world. Forcing all updates to occur in lock step is highly unnatural and has potential fidelity and/or performance problems. It also complicates the models when asynchronous interactions occur.

While some time-stepped systems support multiple time steps, this approach introduces severe limitations because it still forces models to update their state in a synchronous manner. From a more general perspective, time-stepped simulations with multiple time steps are really trying to become discrete event simulations.

**Second**, time-stepped approaches are always concerned with choosing the right step size. Small steps provide more accurate results because they support tighter interactions in time, but they run slower because more steps are required to advance in time. Large step sizes run faster because they require fewer steps, but they often produce less accurate results. The time step must be chosen to support the tightest possible interactions, even when other models do not require such tight time scales.

An example illustrating limitations in time-stepped simulations is in **Computational Fluid Dynamics (CFD)** models of systems containing large numbers of particles that also collide. While it may be possible to construct...
accurate fluid models using a time-stepped approach, the asynchronous particle collisions really demand a discrete-event infrastructure. Discrete-event simulations can support time-stepped event scheduling. However, it is much more difficult to support asynchronous discrete-event scheduling within time-stepped simulations.

The discrete-event approach allows events in the modeled system to be scheduled as they naturally occur in the real world. Skilled modelers know how to significantly reduce the amount of computations performed by their models by only scheduling events when necessary. Discrete event simulations place no restrictions on how tightly in time events can be scheduled. So, the best of both worlds, performance and fidelity, is supported by the discrete-event approach.

Furthermore, powerful asynchronous interrupt constructs can be provided through process-model capabilities that can be extended to pure discrete-event implementations. Process model constructs can significantly reduce the number of lines of code required to develop models by allowing events to arbitrarily pass time before completing their function. This powerful paradigm results in lowering the cost of developing robust models. Process model constructs are illustrated later in this paper.

Events can be scheduled for the same model (i.e., a self-scheduling event or process), between different models residing within the same entity (i.e., a local event), between entities residing on the same processor or on different processors (i.e., autonomous, simulation object, or polymorphic events), or between different simulations operating in a distributed federation (i.e., HLA-style attribute update/reflections and interactions). These event types are discussed in more detail later in this paper.

Publish/subscribe event scheduling mechanisms provide an important abstraction because they do not require specifying recipients of scheduled events. Instead, the simulation engine and/or run time infrastructure delivers published events to subscribing models, components, entities, and/or simulations operating within a distributed environment. Advanced publish/subscribe event delivery mechanisms [33,43] based on interest management techniques determine which simulations, nodes, entities, components, and models receive events based on the actual contents of the event messages. A good example of this data distribution approach is range-based filtering techniques that limit the distribution of federation objects and attribute updates to only those entities containing sensors that are within a specified field of view.

In terms of interoperability and reuse, the fundamental problem is that most discrete event simulations are implemented as active models, which makes them inherently (a) non-portable to other simulation engines and (b) non-reusable because they directly interact with other models.

OSAMS remedies these problems by using (1) hierarchical composition entity and model construction technologies, (2) standardizing the event-scheduling interfaces in a standard modeling framework, (3) providing abstract polymorphic method interfaces to allow hierarchical models to interact while remaining decoupled, (4) providing distributed object technologies to support high performance parallel and distributed computing capabilities and operation in federated or enterprise environments, and (5) supporting consolidated distributed trace file generation with generic data logging capabilities that support testing, debugging, analysis, and VV&A. In addition, these capabilities have also been focused in their design to support and automate interoperability with mainstream standards such as HLA, DIS, TENA, and emerging web-based SOA technologies.

The five critical technology areas in OSAMS are discussed in more detail in the following sections.

### 3.1 Hierarchical Composition Structure

Hierarchical composition structures naturally exist in real world systems. Things are physically composed of other things, which in turn are composed of yet other things, etc. In the physical sciences, this is easily illustrated from macro-to-micro perspectives. From the macro perspective, the universe is composed of galaxies, which are composed of star systems, which are composed of planet systems, which are composed of a planet and its moons, etc. From the micro perspective, molecules are composed of atoms, which are composed of electrons, neutrons, and protons. The neutrons and protons in the nucleus are composed of quarks that are held together by gluons...

In a more practical example involving military defense systems, an aircraft is composed of sensors, various weapon systems, communication devices, a pilot with intelligent behavior, wings and fuselage, one or more jet engines, etc. Each of these physical components can be hierarchically decomposed into additional components and their subcomponents, etc. For example, a radar sensor is typically composed of a transmitter, receiver, power supply, signal processor, data processor, and timing and control subsystems. Each of these subsystems can be further decomposed into more detailed subsystems, etc. In this manner, any level of detail can be modeled.

#### 3.1.1 Composition: Hardware Perspective

It is helpful to understand how simulations and models are composed to computers in a distributed network environment before discussing model composability.
At the top of the hierarchy, Federations are composed of networked federates that communicate through a run time infrastructure. Typical communication overheads are in the millisecond range for federate-to-federate interactions. Overheads are sometimes even larger when operating across a wide area network or when time management is required to support large federations. Federates (i.e., individual simulations) are composed of one or more machines that potentially communicate with overheads in the tenth of a millisecond range. Machines are composed of one or more processing nodes that can communicate through shared memory or other high-speed local interprocess communications mechanisms in the microsecond range. Nodes are composed of one or more threads that have context switching overheads in the nanosecond range. Finally, threads can be composed of objects and functions that communicate through direct function calls or methods in less than a nanosecond.

Understanding these interprocess communication mechanisms and their performance overheads is essential when composing a simulation in a distributed computing environment. For example, the distributed simulation will not perform well if highly interactive models are placed in different federates where the communication overheads are very large. The distributed simulation would perform much better if the highly interacting models were placed within the same computer, processing node, or thread if possible. The techniques described in this paper provide complete flexibility in composing software models to machines on system architectures ranging from highly mobile personal computers to networks of supercomputers (and everything in between).

### 3.1.2 Composition: Model Perspective

In military simulations, federations are composed of simulations (i.e., federates) that are composed of entities that are composed of model-components that are further composed of sub-components, etc.

In practice, each federate normally provides a specific functionality (e.g., collections of models for: land, air, sea, space, communications, protective materials, etc.). In theory, the federation should be capable of supporting all of the systems spanned by the functionality of its participating federates. Participating federation members are normally selected to provide all of the necessary functionality required to conduct an exercise.

Integration challenges arise when federates overlap in functionality and/or when critical gaps in functionality are identified that are not provided by any of the member federates. Software developers must disable models residing within a federate that are provided by other federates, while also creating new models to fill capability gaps. Agreement on exchanged data, interaction patterns, and message formats can pose additional challenges. Federates communicate with one another through a run time infrastructure. HLA Interoperability rules forbid backdoor communications between federates because this would bypass the interoperability mechanisms and therefore limit the potential for interoperability and reuse.

This paper does not recommend developing next generation models as standalone federates, but instead as interoperable models that can be composed into any logical composition structure and hardware mapping desired by the operator. OSAMS does not replace HLA, but instead complements it by providing interoperability standards to the models themselves. OSAMS allows developers to build interoperable models rather than building, patching, and integrating disjoint simulations.

#### 3.1.2.1 Composition of Federates into Entities and Component Models

Federates are composed of entities such as ships, aircraft, and ground units. Entities usually represent semi-self-contained and potentially moving systems. Sometimes, the word platform or simulation object is also used to describe an entity.\(^3\) Formally, an entity is a special kind of simulation object that can also publish and subscribe to federation objects (see discussion on Federation Object Technologies). Simulation objects can be distributed to different processing nodes to support parallel and distributed event processing.

Sometimes, entities are modeled as aggregate systems. For example, an aircraft carrier might represent an entity that models all of its aircraft as components, even when they are in flight and separated from the carrier. Other examples of aggregation might include treating air missions involving multiple aircraft as a single mission entity. Ground units can be aggregated at different levels ranging from single warfighters to fighter units, squads, platoons, companies, battalions, or divisions. The level of model aggregation can play a significant role in supporting interoperability, especially when sensor models expect to detect individual battlefield objects that are modeled as aggregates. Choosing the right level of aggregation and/or supporting mixed-level resolution strategies must be considered when developing highly interoperable models.

Model ontology standards, such as the Base Object Model (BOM) [1][40], offer a formal way to characterize interfaces between models and the roles they are expected to play with other models. Ontology-based technologies may play a significant role in establishing rules for which

\(^3\) For simplicity, this paper uses the terms entity, platform, and simulation object interchangeably.
models can “play” with which other models. Such technologies may also assist during VV&A to provide traceability from execution, to the system design, to the conceptual design, and ultimately to system requirements.

The functionality of an entity is usually provided in its component models. The entity itself does not normally provide behaviors, but rather is a container of interacting component models. One important consideration is that when running in a parallel, distributed, and/or federated environment, the internal components of an entity are always located on the same processor. Entities may be distributed to different processors within a federate or even within different federates, but their internal model components are not. This has significant impact on the rules for interoperability and reuse because the internal components within an entity do not require a message-passing infrastructure to mutually interact.

From a modeler’s perspective, entities must never be permitted to directly invoke methods on other entities. Message passing abstractions or distributed object capabilities must be used to support data exchanges and interactions between entities. This is discussed later in the section on Distributed Object Technologies.

Entities are composed of components that can be recursively composed of other components, etc. So, an aircraft might be composed of a radar system, a visual sensor, behavior models, motion models, and weapon systems (see Figure 3).

To support flexible and dynamically reconfigurable component structures, entities and model components must not hard-code their organizational structure. Instead, a text-based file with a standard format, or alternatively a database interface, might be used to specify the composition structure of an entity and its components along with all of the model parameters that are required to initialize each component. A graphical front-end user interface may assist in defining entity/component structures, critical model behavior parameters, and scenario generation specifications that define each entity instance along with their individual mission plans, capability parameters, and objectives. Missions can be modeled as a sequence of state machines implemented as model components. Rules for transitioning can be established at run time through dynamically triggered rule sets that generate state-changing interrupts.

To support complete flexibility, it is important for entities to have the capability of dynamically reconfiguring their component structures during execution. For example, a high fidelity model could be swapped out and replaced by a lower fidelity model when such precision is not needed, but later swapped back in when the situation demands results with higher fidelity. The visualization community has used this technique when viewing images from different distance perspectives. As the user zooms in, the image resolution increases. Dynamic model composition can reduce processing overheads, thus resulting in faster run times. However, the model swapping transitions can be complex, especially in those cases where the internal state of the model must be reinitialized from a model with a different aggregate state representation.

3.2 Standard Modeling Framework

A standard modeling framework provides all of the interfaces necessary for models to be constructed, destructed, and to schedule/process events. It also provides cognitive processing capabilities.

An event is scheduled to occur at a specific point in time. Usually events are implemented as a method or function that is activated at the time of the event. Events potentially modify state, record output data to disk or to external systems such as graphical interfaces and databases, and/or schedule new events. Sophisticated process model extensions allow events to persist over time and to be interruptible by other events or processes as conditions change. Process model constructs can significantly reduce the amount of software lines of code necessary to develop highly interacting complex models.
A sensitivity list mechanism allows methods on arbitrary objects to specify a set of variables that when modified will trigger an invocation of the method. While being processed, the method may modify other variables that trigger the invocation of other sensitive methods. This capability provides a foundation for many cognitive algorithms used to represent human behavior. Examples include neural networks, genetic algorithms, and expert or rule-based systems. The sensitivity list can also trigger process model interrupts for conditions involving multiple variables. In this manner, rules can be established that are automatically checked (and possibly trigger other rules) whenever specified values change. The sensitivity list can trigger state transitions representing a series of planned mission-oriented activities.

3.2.1 Events

Four basic types of events have been defined in sequential, parallel, and distributed simulation systems. Sophisticated simulation engines allow modelers to specify actual data arguments and object methods in supporting events. More primitive engines simply store event arguments in messages that are passed to generic functions or methods. Such messages can be packed in variable-length buffers that store name value pairs (e.g., the HLA approach) or simply as fixed-length message structures (e.g., the DIS approach).

In some simulation infrastructures such as the HLA runtime infrastructure, events are undirected, meaning that the recipient of the event is unknown by the scheduler. Instead, publish and subscribe mechanisms deliver the event to all subscribers who meet the subscription requirements. However, in normal discrete event object-oriented systems, object handles provide a useful abstraction for identifying a specific recipient of a directed event. In a parallel simulation, an object handle might consist of three integer values that specify the node, object type, and local identifier of the simulation object. In some cases (see the discussion on local events provided below) simple pointer references can be used to specify the event recipient.

The four basic event types have been implemented in several systems and are illustrated in Figure 4.

![Figure 4: Four basic event types.](image-url)
1. **Autonomous Events** are implemented as active objects that act on passive simulation objects. Autonomous events are similar to autonomous mobile agents [30], except that they are only active at a discrete point in time. They do not persist beyond the time of their event. Autonomous events are useful in special circumstances where it is important to maintain a clear separation between the event and the simulation object. The event recipient is designated during scheduling by an abstract object handle.

2. **Simulation Object Events** are implemented as methods on simulation objects. Unlike autonomous events, simulation object events directly schedule a specified method implemented by the simulation object. Simulation objects are permitted to schedule events for other simulation objects using either the Autonomous event or SimObj event mechanisms. Like autonomous events, the simulation object event recipient is designated during scheduling by an abstract object handle.

3. **Local Events** are implemented as methods on any object contained within the simulation object or its internal components. Instead of using abstract object handles to identify the receiving simulation object for the event, it is assumed that the local event is scheduled for the same simulation object. A simple pointer or reference to the object implementing the local event determines which internal object receives the event. Simulation objects are never permitted to schedule local events for other simulation objects because this would violate fundamental object-oriented encapsulation principles. A simulation object should never have access to the pointer of an encapsulated object contained by another simulation object. Local events provide a natural mechanism for scheduling events between objects within a simulation object. They are also used to support process model capabilities. Processes are always implemented as local events.

4. **Polymorphic Events** are implemented as an abstract event-scheduling interface with event handling methods that are dynamically registered (or unregistered) and potentially reconfigured during run time by component models. Polymorphic events are similar to polymorphic methods except that they are scheduled in time instead of directly invoked as function calls. Polymorphic events can help decouple software components, while providing a foundation for supporting mixed resolution models. Polymorphic events trigger zero or more registered handlers when they are processed. Polymorphic events can be scheduled for different simulation objects or for the same simulation object. Polymorphic events can also be scheduled for individual components within an entity.

While these four event types are extremely useful, especially in implementing core distributed object services in simulation engines, in practice only local events and possibly polymorphic events scheduled for local components support the essential interoperability mechanisms prescribed in OSAMS. This is because of the important consideration that interacting entities may reside in different federates. So, while powerful simulation engines might internally use all of these event types to provide their core services (e.g., distributed object capabilities), interoperable component models should only use local event services or polymorphic events for local components. Entities must interact with other entities through the interaction and federation object capabilities that are provided by the distributed object services specified by OSAMS. Otherwise, HLA interoperability and operation on multiprocessor machines will be impossible to support in an automated manner.

### 3.2.2 Dynamic Simulation Object Construction/Destruction

Interfaces for dynamic simulation object construction and destruction must be specified and supported by OSAMS-compliant simulation engines to allow new entities to be created or destroyed during the course of the simulation. Event scheduling techniques must be used to support these services because object construction and deconstruction occurs in simulated time for simulation objects potentially residing on other processors and/or federates. Various approaches have been explored in previous simulation engines including: (1) different versions of constructor and/or initialization method arguments, and (2) model creation interactions that pack the composability structure, initial state values, and federation objects to be published along with the entity construction event. One important consideration is that new objects must only be created within simulations that can actually create the objects. It does no good to schedule the creation of an aircraft entity within a simulation that is only capable of modeling ground units or ships and submarines. Model clustering that assigns model types to compute nodes must be determined by the simulation engine to determine which models can be created within each simulation and potentially within the nodes of a parallel/distributed OSAMS simulation that itself might be composed of several different simulations.

---

4 Any user-defined object contained within the Simulation Object or any of its hierarchical components can implement a local event.
3.2.3 Process Model Constructs

Any event can become a process through the use of a few simple macros. A process is nothing more than an event that passes simulated time before exiting the event method. Some processes persist for the entire duration of the simulation execution. Sometimes, processes are called persistent events because they persist over a period of simulated time. One of the benefits of using processes over events is that the context within the event method is preserved when waking up from a wait statement. All of the important local variables are preserved. An example of a very simple process that loops ten times with a wait statement is shown below.

**Code Segment 2:** This code segment provides an example of a process with a loop and simple wait statement. More complex processes can be constructed with multiple wait statements that wake up when specific conditions, interrupts, or timeouts occur. When processes wake up, simulated time may have advanced. All local variables defined by the P_LV macro are restored to the previous values that they had prior to the wait statement.

```c
void S_MySimObj::WaitProcess(int a) {
    double b = 0;
    P_VAR
    P_LV(int, i)
    P_LV(double, c)
    P_BEGIN(1)
    c = 0.0;
    for (i=0; i<10; i++) {
        a += 1;
        b += 1.0;
        c += 1.0;
        WAIT(1, 10.0)
    }
    P_END
}
```

Processes can have several ways of passing time. In the simplest form, processes can wait a specified amount of time using the WAIT command. However, processes can also wait for various types of semaphore variables to be set or change state using the WAIT_FOR command. Several processes may wake up when a shared semaphore is set. A timeout can be provided to wake processes up after a specified amount of time has elapsed, even if the semaphore has not been set. The following semaphore types are typically supported in process models:

1. **Logical semaphore** - can have values 0 or 1. Processes waiting on a logical semaphore wake up when the semaphore is set to 1. Usually the semaphore represents some condition that is required for the process to continue its activity.

2. **Integer (or counter) semaphore** - allows processes to wake up when the integer (or counter) value is non-zero. This is useful when a model needs to wait for work to arrive before processing. A good example of this is a sensor waiting for at least one target to be in range to produce detections.

3. **Double semaphore** - allows processes to wake up when the double-precision value is non-zero. This is useful for supporting process activities that depend on a continuous threshold variable.

Processes can also wait for specified interrupts to occur. A timeout may be provided to allow the process to wake up after an allotted time when interrupts do not occur within the specified time. Interrupts are enumerated and stored within a fixed sized integer bitfield. Processes can selectively mask interrupts that are of interest. Waits on semaphores can be interrupted. An example might be a sensor waiting on a counter semaphore for one or more objects to enter its field of view. While waiting, an interrupt might occur indicating that the sensor has been hit by weapon fire and is now destroyed. The sensor process should handle the interrupt by terminating.

The *resource* data type works with the process model to allow processes to wait until a specified amount of a requested resource becomes available. This is extremely useful in modeling queuing systems where customers wait to be served by a limited set of server resources. Processes waiting for resources to become available are prioritized according to a selected queuing discipline. Examples are First-In First-Out (FIFO), Last-In First-Out (LIFO), by amount of the resource requested (high or low can be chosen to represent the top priority), and arbitrary priority assignments that are assigned by the model itself. The following resource types are commonly supported:

1. **Integer resources** - used to represent numbers of resources. An example might be ammunition that needs to be loaded in a weapon system before the model can continue with its mission.

2. **Double (floating point) resources** - used to represent an amount of a continuous resource. An example might be the amount of fuel that must be pumped into an aircraft before it is permitted to leave its airbase.

3.2.4 Sensitivity List

The modeling framework provides a sensitivity list mechanism that automatically invokes registered methods when specified attributes are modified. This capability can be extended to processes to allow them to wake up from process-model WAIT statements when arbitrary complex expressions involving several attributes are
satisfied. Because the invoked functions themselves are allowed to modify other variables in sensitivity lists, a powerful capability is provided to support cognitive algorithms such as neural networks, genetic algorithms, and rule-based logic in expert systems. These services are essential for developing reusable Human Behavior Representation (HBR) components.

The sensitivity list mechanism allows methods to individually assign priorities to each registered variable. This coordinates the ordering of method invocations, which can be critical when evaluating rules efficiently or obtaining convergence for numerical problems. To speed up convergence for numerical problems, the sensitivity list also allows tolerances to be specified on floating point values. Only values that change by more than the specified tolerance trigger the invocation of registered methods. Finally, the sensitivity list mechanism terminates either when there are no more methods to invoke, or when a user-specified upper limit for the maximum number of methods invoked is reached.

3.3 Abstract Polymorphic Methods

Entities are internally decomposed into model components that can be further decomposed into more specific subcomponents, etc. This provides a rich hierarchically managed recursive infrastructure for decomposing model functionality. Polymorphic methods with component-based scope resolution provide a powerful mechanism to remove dependencies between components while promoting their interoperability and reuse.

Like traditional callback systems, a component can invoke a polymorphic function that in turn activates polymorphic methods that were registered by other components. This is very similar to familiar general-purpose GUI callback systems that allow applications to register handlers when buttons are mouse-clicked, etc., except that the polymorphic method system is object-oriented. Also, the polymorphic method system allows zero or more methods to be registered and invoked when the interface is called.

The hierarchical polymorphic method infrastructure manages which methods have been registered by which components. The invoker of a polymorphic function does not know which components have registered their polymorphic methods, nor does the invoker know which polymorphic methods are applied by registering components when activated. This double abstraction barrier has been borrowed from other highly successful interoperability technologies such as CORBA, HLA, and web-based SOA technologies using XML [49], Schemas [50], SOAP [51], WSDL [48,54], DOM [52], and OWL [53]. The double abstraction barrier methodology facilitates the development of interoperable components that can be reused in different applications because there are no hard-coded object dependencies.

The polymorphic method system is much more powerful than the traditional old-school object-oriented inheritance and virtual function approach to polymorphism. This approach does not require inheritance or virtual functions to achieve polymorphism. Instead, a special macro is used to define the polymorphic interface. This macro generates a new polymorphic function with the specified interface that can be invoked to activate all corresponding polymorphic methods that are in scope and have been registered. Note that the invoker of the polymorphic function only references the interface and has no knowledge of how the interface is implemented or which classes are used to implement the interface. Thus, models are allowed to interact without requiring hard-coded class dependencies. Only the agreed upon interface is required.

Through another macro, any object may define one or more of its methods to correspond to the polymorphic interface. Again, this demonstrates the powerful double abstraction barrier principle. A conceptual example of polymorphic methods used for sending sensor detections to a track fusion model is shown in Figure 5.

![Figure 5: An example of two components interacting through a polymorphic method.](image)

In this example, a Radar component on a Ship entity sends detections to the Track Fusion component through the polymorphic method mechanism. The radar component generates detections that are processed by the Track Fusion component when invoking the Process Detections polymorphic function. This in turn activates the Fuse Detections method in the track fusion component that has been registered as a polymorphic Process Detections method. The double-abstraction barrier

5 No inheritance is required and the object can be named anything.

6 The method can also have any name as long as the signature matches the defined interface.
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.

Scope resolution is necessary to localize/bound which components are able to process the polymorphic method. Models can specify a component in the hierarchy when invoking a polymorphic function. Only registered methods at that component level or below in the hierarchy will be activated. For example, this allows multiple sensor systems within an entity to use polymorphic methods to coordinate their internal model interactions without spilling over into other sensor systems. To maximize interoperability and reuse, models are allowed to access their current, parent, grandparent, etc., components, but they should never directly access peer or child components because that would imply improper knowledge of the dynamic composition structure. As a general rule, specific methods on components should never be invoked by other components. All interactions between different model components should occur through polymorphic methods.

3.4 Distributed Object Technologies

The distributed object technologies mirror HLA functionality between entities and components with much simpler and automated interfaces that support Federation Objects, Interactions, Interest Management, and Ownership Management [44].

Entities and their components contain attributes that are mapped into dynamically created or destroyed federation objects during the execution of the simulation. Federation objects are designed to automate integration with HLA because they are similar to the Declaration Management and Object Management services.

To simplify their use, the attributes themselves are represented as special exportable classes that provide useful data types. These are shown in Figure 6.

Through operator overloading, these attribute objects behave as their primitive data types. However, the distribution of their new values to subscribing entities and components is automated whenever they are modified. Subscribers are notified through registered polymorphic methods when federation objects are discovered, removed, and when their attributes are modified. These attribute data types can be single valued or they can be used as arrays. The cardinality of each data type is specified in a federation object input file. Run time error checking is performed to ensure that the cardinalities between the input file and the implementation agree.

Figure 6: Exportable attribute types supported by federation objects.

One of the most interesting attribute representations is the dynamic data type, which is implemented as a list of object segments that provide methods as a function of time to compute their value. Each segment has a start and end time to characterize its period of validity. Multiple segments can be inserted into dynamic data types to represent a list of complex time-based computed values. Users simply request the attribute value at a given time. The dynamic attribute finds the corresponding segment in its list that bound the requested time before computing the dynamic value.

Dynamic attributes are provided for integer, double, logical, and position data types. In the case of dynamic doubles and position, a wide variety of built-in segment types are provided. The infrastructure is extensible to support additional user-defined segment types. A coordinate system transformation library is also provided to assist in data translations between Earth Central Inertial (ECI), Earth Central Rotating (ECR), World Geological Survey 84 (WGS84) ellipsoidal representations, and Round Earth coordinate systems. All of the motion algorithms are designed to work correctly in each of the supported coordinate systems.

Figure 7 shows the relationship between various classes implementing entities, components and federation objects. Logical processes provide the base class that simulation objects inherit from. To cleanly separate event management services from modeling framework services, the simulation engine coordinates generic event processing using the logical process base class. The simulation object class inherits from the logical process base class and provides component creation interfaces for model developers. The composite simulation object class automates the hierarchical component construction of simulation objects using run time class input files.\(^7\)

---

\(^7\) Run time class input files are conceptually similar to XML documents and Schemas. They provide object structure,
specify entity and component structures. The entity class provides the foundation for developing entity models. It contains the root federation object manager for the entity.

![Entities & Components](image)

**Figure 7:** Composition of entities, components, and their federation objects.

The federation object manager provides interfaces to create, publish, and subscribe to federation objects. It also provides many useful access methods to obtain federation objects by name, Id, type, etc. In addition, the federation object manager provides iterator methods to loop over all received federation objects of specific types. Like components, federation object managers are linked in a hierarchical fashion to coordinate subscriptions and federation object discoveries between components.

Simulation objects may contain components. A composite component uses runtime class input files to initialize internal attributes and to automatically construct children subcomponents. Like the entity class, the entity component also contains a federation object manager. It is linked to its parent federation object manager in the same tree-like manner as its parent component or entity. Applications provide their models by inheriting from the entity, entity component, and federation object classes. These user-defined classes may contain other classes without any inheritance or naming requirements.

This hierarchical structure provides a very powerful way to develop highly interoperable models that share exportable state variables with other models through publish and subscribe mechanisms. Publish and subscribe mechanisms are provided between entities and between the components within an entity. Interest management capabilities allow subscribers to specify conditions on attributes of the federation objects they discover. The most common example of interest management is range-based filtering where only published objects located within a specified distance of an entity are discovered. This capability can be extremely challenging to provide in an efficient manner because objects can move with arbitrary and sometimes unpredictable motion.

A brute-force range-filtering algorithm would compare every object with every other object periodically to determine which objects are in range with each other. However, the brute force approach does not scale when the number of entities becomes large and when operating in a parallel or distributed environment. Other approaches compute and schedule events when objects move in and out of range. However, this approach can thrash when objects unexpectedly change their motion. It also requires \( n^2 \) computations and consumes \( n^2 \) memory resources to hold scheduled events. More sophisticated approaches use advanced data structures such as multi-resolution hierarchical grids to determine publication/subscription overlaps in a scalable manner. Computations and events are only scheduled when objects are close to entering or exiting the range of other objects. Hierarchical grids have been shown to improve brute force performance of large systems by orders of magnitude. Even larger gains are possible when compared to the thrashing that often occurs with exact enter/exit range computations.

It is critical for the distributed object service to provide efficient and scalable interest management for subscribing systems with different resolutions. Without efficient interest management, performance breaks down quickly when the numbers of federation objects gets large. Interest management algorithms must support multiple resolutions, especially when providing range-based filtering. When executing in a multiprocessing environment, the interest management computations should also be distributed to avoid computational and message-passing bottlenecks. Efficient multicast techniques to distribute the filtered data through shared memory and/or standard networks are necessary to reduce message-passing overheads in large interconnected computing systems.

### 3.5 Consolidated Trace File Generation and Data Logging

Providing software testing, debugging, analysis, and VV&A of complex distributed simulations is not easy. Without standards, the process of instrumenting legacy simulations to generate event traces with data logs that can be analyzed for correctness can be tedious and difficult to obtain. While HLA could be used to automatically collect event information exchanged between federates, it is not set up to collect internal event processing data within each simulation.
There are no common interfaces or widely used output data formats for instrumenting such systems. Furthermore, there are no standard tools for producing and analyzing system traces. Typical software development testing strategies involve generating debug output in textual form, analyzing performance profiles, creating trace files of system calls and/or stack histories, “sniffing” the network for messages, quantifying the test coverage, and monitoring memory references and allocation/deallocation operations for incorrect usage. However, there are very few tools that actually produce meaningful trace files indicating time-based sequences of interactions between core software components.

Some simulation engines and database infrastructures may automatically generate trace files. However, this capability is usually not automated with legacy simulations. OSAMS leverages existing simulation trace file generation technologies, while extending the technologies further to support generic data logging and analysis capabilities.

Software components requiring rigorous testing can vary dramatically in granularity, ranging from distributed systems of systems operating in the GIG down to the smallest subroutines or object methods within a single executable application. Trace files are crucial in understanding, verifying, and validating complex systems that interact in a meaningful dependency time line. Trace files are also invaluable in supporting After Action Review (AAR) of complex scenarios. In some cases, it is advantageous to store large and complex trace file records in a database such as Access, MySQL, or Oracle. In most cases, however, a flat trace file is adequate.

Using the Unified Modeling Language (UML) methodology [12], trace files can be used to validate sequence diagrams, state diagrams, activity diagrams, and use-case diagrams. UML representations involving time-based scheduling semantics have not been standardized and widely accepted by the broad community. In the context of the DoD Architecture Framework (DoDAF) [8,57], testing time lines can help to verify and validate both the system and operational views of an application’s actual operation. Other commonly used forms of representing data exchange models include: Web Service Definition Language (WSDL), Base Object Models (BOM), XML Schema, Joint Consultation Command and Control Information Exchange Data Model (JC3IEDM) [25], etc. In all of these cases, both the data exchanged and the interplay between the software components are documented in the conceptual and software implementation designs.

Rigorously and cost-effectively testing model components operating within a more complex distributed simulation environment is a significant challenge that would strongly benefit from a standard such as OSAMS. Such a capability would help both our economy and our national priority to protect the Homeland. However, until research can make software testing more thorough and cost effective, testing would benefit from empirical approaches based on understanding correspondences among instrumented observables. A user-guided graphical visualization tool to view the time lines reconstructed by trace files produced during execution would help immensely. This capability is normally provided through offline analysis. However, OSAMS could also support real-time analysis of trace files to assist in testing while the system is actually executing in a live operational setting.

OSAMS addresses these concerns by providing a standard framework that can automate trace file generation in a standard format such as XML or other text-readable formats.

### 3.5.1 Trace File Generation

Distributed simulation engines often generate ASCII-based human-readable trace files (when enabled) to allow their applications to trace time-tagged event scheduling interactions between objects and to provide additional information that is helpful for debugging, validating, and verifying software correctness. Formats are typically chosen to promote readability by humans, but could very easily be generated in other formats such as XML to support web-based service-oriented standards. Records in the trace file might contain the following information for each event:

1. Simulation time of the event
2. Name of the event
3. Object type and instance processing the event
4. Reentry label for processes
5. List of scheduled events
6. CPU usage (optional)
7. User-defined output messages
8. Logged data (type, name, value)

Note that some of the fields are optional and may be enabled/disabled at run time through a command-line option, an environment variable, or through a configuration file. The reason certain fields are optional is that they either involve: (1) information which would not be repeatable, such as the exact time it took to process an event or the specific flow-control mechanism used to send a message, or (2) information which may or may not be important to the user.

Data logging can be captured in the trace file and then later analyzed to support testing, debugging, analysis, data mining, and validation.
### 3.5.2 Time Line Analysis Tool

User-data logged in trace files can be visualized, mined, and analyzed within a standardized graphical Time Line Analysis Tool. Such a tool would greatly benefit testing, debugging, analysis, and VV&A of simulation models.

Filtering mechanisms would allow users to quickly filter out model interactions that are not of interest. Simple data mining filters might be based on object types, object instances, event types, and the tracking of specific transactions. For example, one might want to visualize all operations generated by a specific object instance as it begins a specified transaction. A conceptual graphical interface for the Time Line Analysis Tool is shown in Figure 8.

**Figure 8**: Conceptual Time Line Analysis Tool.

By either mousing or double clicking over one of the circles on a time line, the entire record of the event may be displayed. Users may zoom in or out to explore detailed interactions between objects (or components) in the system. Various analysis options are provided to support correlations and statistical analysis of the execution. The goal of this tool is to allow users to quickly filter out unwanted data to clearly visualize and analyze their software behavior. All of this is focused on reducing the cost of testing, debugging, analyzing, and validating the models as they operate in a distributed environment.

### 3.5.3 Additional Data Logging Capabilities

In addition to trace file generation and data logging, OSAMS implementations should be able to provide additional test, debug, and monitoring capabilities. These capabilities are briefly listed here.

1. Memory management diagnostic tools that operate within a persistence framework with memory checking services that detect leaks or corrupted memory segments
2. Event processing statistics that summarize how many events of each type were processed and how much time was spent in each type of event
3. Message passing statistics that record the number of messages sent by type, bandwidth, and their latencies
4. Critical path analysis that shows where the bottlenecks are along with the maximum speedup possible within the distributed simulation system

### 4 Cost Saving Benefits of OSAMS

The purpose of this section is to identify and characterize how the development of an Open System Architecture for Modeling and Simulation (OSAMS) will affect long-term costs across multiple programs [10,16]. While this objective is very difficult to accomplish without actually comparing the costs of identical systems developed with and without OSAMS, this section will attempt to articulate expected cost savings based on fundamental principles, common sense, experience with other programs, existing theoretical cost models, and on limited information whenever available. While OSAMS does require an initial cost to develop and maintain, this cost is small compared to the potential savings that can be provided. Furthermore, these costs can be shared across multiple programs and industry.

#### 4.1 Software Development of New Models

Developing new models that are compliant with OSAMS has the potential to reduce software costs because OSAMS provides a rich modeling framework and suite of utilities designed to reduce the number of lines of code necessary to implement models. This results in cleaner, robust, and more computationally efficient software implementations.

Examples of these savings from previous programs have shown the proposed OSAMS modeling framework and utility suite to save between a factor of three and five for models of complex systems. At the cost of $100 per line of code suggested by the COCOMO model [21], these projected savings will be substantial for moderate-to-large models, especially those with complicated logic and/or rule sets.
4.2 Maintenance of Existing Model Components

Instead of maintaining large complex simulations containing collections of interacting models, OSAMS proposes to establish repositories (or libraries) of encapsulated OSAMS-compliant models. These models are self-contained, well documented (perhaps using the BOM ontology), and interoperable with other models through abstract polymorphic interfaces. This has tremendous cost-saving benefits. Instead of maintaining large complex software simulation systems that are hard to navigate, test, and modify, OSAMS allows each model to be maintained as a bite-sized fully encapsulated module designed to interoperate with other models in a standard architecture.

Code maintenance of legacy simulations can be extremely difficult, especially when the number of lines of code gets large (i.e., millions of lines of code). To make a change to the system, developers must know which files require modification and how such modifications will affect the rest of the system. Making complex model changes can be difficult enough for the original software developers who are intimate with the design and implementation of the legacy system. Software developers who are not familiar with the design of large complex legacy systems have a nearly impossible task of maintaining or modifying the system. Any change is likely to have side affects that will break the system. Multiple changes (or worse, messy workarounds and/or patches accumulated to the code over time) almost always wreak havoc with the overall structure of the system making the resulting software structure brittle and overly expensive to maintain. This undisciplined way of maintaining software systems ultimately has the affect of “mothballing” large software investments when the original engineering team moves on to other programs and is no longer available to maintain the software.

One of the benefits of OSAMS is that the maintenance of models becomes cost effective for several reasons. First, model implementations are completely encapsulated from other model implementations. This means that dependencies are minimized. Code changes only affect the internals of a model and potentially its outputs. Changes should not break other models. Second, because of the encapsulation of models into components, the models themselves remain small enough in terms of code size to navigate and maintain by software engineers who potentially were not the original developers. Third, OSAMS models are developed using a standard modeling framework. So, software engineers familiar with the OSAMS interfaces will be able to quickly understand and work with the event-scheduling semantics of the models. It also is a significant benefit that the models themselves will require fewer lines of code to develop and maintain.

All of the reasons described above combine synergistically to provide a scalable software engineering architecture and methodology for maintaining or modifying software components. Scalable software engineering fundamentally means that the software complexity of the system grows linearly with the number of lines of code. Fully encapsulated and decoupled component models in OSAMS provide this characteristic. On the other hand, spaghetti software designs (or patched systems) become unwieldy and eventually unmaintainable because of their characteristic $n^2$ couplings between the software components (i.e., everything potentially depends on everything).

4.3 Testing and Troubleshooting Problems

One of the most important aspects of software development is testing and troubleshooting. It is nearly impossible to troubleshoot a large complex system with many software dependencies and assumptions, especially if they do not have an organized test strategy. Because OSAMS models are completely encapsulated and decoupled from one another, it is possible to develop robust standalone unit test suites. This means that modifications can be made to a model and then unit tested in isolation from the rest of the system. Problems can be quickly identified and resolved without wasting lots of time hunting down the cause. This is not possible in systems where the models are highly coupled with hidden assumptions and dependencies. Troubleshooting such systems can be extremely slow and painful (i.e., expensive) because it is not always obvious where the problem originated, and how to fix it without breaking other parts of the code.

Furthermore, additional interface and self-consistency tests can be built into the simulation engine implementing OSAMS to (1) find discrepancies, (2) record or analyze interaction timeline traces, (3) provide diagnostics, etc. during the operation of the models. The component model techniques in OSAMS are designed to reduce the time required to close reported problems with the system. Of course, this depends on a fully instrumented robust OSAMS-compliant simulation engine and trace file analysis tools.

4.4 Modeling and Simulation Training for Software Developers

One of the significant cost factors in modeling and simulation is the learning curve required for software developers to become productive on a particular program.
Because there are currently no widely accepted standards for how models are developed, each simulation program has its own learning curve to understand its architecture and modeling framework. This can be very expensive and time consuming on large complex programs, leading to poor development and maintenance productivity.

OSAMS solves this problem by standardizing how models are developed. The National Training Services Association (NTSA) [34] has developed a Modeling and Simulation Professional Certification Commission (M&SPCC) [32] to certify professionals in M&S. A certification program for OSAMS would train and identify M&S professionals who are proficient in developing and/or maintaining OSAMS models.

Consolidating training will leverage these costs throughout multiple programs. Engineers will be able to freely work with documented models developed by others because their basic form is standardized. Instead of inefficiently maintaining specialized teams of experts for each simulation program, a consolidated team of OSAMS certified engineers could maintain all OSAMS models, independent of the developing organization. Subject matter experts would still be required when it comes to model behaviors. However, subject matter experts on the design/architecture of particular simulation systems would not be required because OSAMS would provide a common architecture that is familiar to all developers.

4.5 Verifying, Validating, and Accrediting Model Components

One of the most critical, and often expensive, steps in developing useful simulation capabilities is to verify, validate and accredit the models for particular usage and/or specific scenarios. Normally, entire simulations are required to go through the VV&A process [9,36,39]. OSAMS helps significantly reduce VV&A costs by encapsulating model components that can be individually verified and validated. While it is still important to verify and accredit collections of models operating in a particular scenario as a complete system, OSAMS still has the potential of significantly reducing overall VV&A costs. This is because validated OSAMS simulations are constructed of already validated model components.

Furthermore, software modifications to validated OSAMS models simply require revalidating just the modified model, and not the entire collection of models used by the simulation. On the other hand, modifications to legacy simulations that are complex and unwieldy may be very difficult to revalidate, especially when software dependencies between the models is high or difficult to understand. It is hard enough to test and verify that the modifications did not break anything, let alone revalidate the entire simulation system. Trace file generation and data logging capabilities can be used to automate much of the validation process.

Furthermore, by operating in a common OSAMS-compliant environment, standard tools can be provided to help verify the correct operation of models. Such tools can analyze models during composition to ensure that they are truly interoperable, not just in their interfaces, but also in their level of fidelity and the roles they play. Traceability can potentially be provided from (1) requirements to (2) conceptual design to (3) software design to (4) the actual implementation of the system. For example, OSAMS can provide checks to verify at run time that the system is consistent with the UML sequence or state diagrams describing how models were designed to interact with other models in simulated time. Without standards such as OSAMS, timeline analysis, etc., used for verifying the software implementation is nearly impossible.

Another example where costs can be dramatically reduced is through common analysis and data/mining tools that operate on standard OSAMS trace file output. All events can generate timeline output in the form of trace files that can be analyzed by subject matter experts to determine if the models are behaving as expected in the real world. Such tools will streamline the validation process.

4.6 Composing Simulations from Model Components and Legacy Simulations

One of the important benefits of OSAMS is that simulations can be composed from both model components and legacy simulations. OSAMS model components stored in a repository or library will contain metadata describing their inputs and outputs, as well as other pertinent information such as run-time performance, fidelity, expected roles played with other abstract OSAMS models, etc. A composition tool using the metadata from the repository can be developed to ensure that the selected models are truly interoperable. The composition tool can also help users find existing models or identify gaps where new models must be developed.

OSAMS simulations are constructed of Entities, which are hierarchically composed of model components. Through model composition, exercises can be quickly constructed from such components and/or already composed entities.

The distributed object capabilities in OSAMS were designed to allow compliant simulation engines to automate interoperability with other simulations executing as a federation through standards such as HLA and DIS. Using a common OSAMS-compliant simulation engine will isolate and consolidate interface translations from one data model or representation to another [19,37].
The distributed object capabilities in OSAMS were specifically designed to allow such translations to be programmed once and then to occur automatically.

4.7 Constructing and Operating an Exercise or Experiment

One of the most expensive activities involving modeling and simulation is entity/model construction, scenario generation, and operating an exercise. Federations involving multiple simulation tools require operational expertise for each unique system, their input data formats, related graphical displays, and a suite of federation management tools. Furthermore, operating such federations may require management of a network of interconnected computers that host these systems, making it difficult to execute the simulation on a single laptop. It is extremely expensive for operators to learn all of these tools and to set up each system to conduct an exercise. One of the goals of OSAMS is to not require operators to have any special subject matter expertise on particular models or simulation systems. Instead, operators are only required to be familiar with a common suite of OSAMS composability, operation, and analysis tools.

OSAMS reduces this problem by establishing common tools and formats for scenario generation, model construction, and operating an exercise. Operators only have to learn one system that was designed to minimize the effort required to run experiments and/or exercises. Furthermore, simulations constructed solely from OSAMS models are flexible in terms of hardware requirements. OSAMS simulations can run on any combination of PDAs, laptops, networks, and/or supercomputers.

4.8 Analyzing the Results of an Exercise or Experiment

After Action Review (AAR) of a simulation execution is critical. Analysis of engineering simulation experiments provides measures of effectiveness and performance that are used by acquisition to make recommendations. Training simulations provide performance ratings or scores that are helpful in assessing human performance and/or decision-making. Campaign simulations help establish CONOPS, while also helping assess and predict future outcomes in real time. Models can sometimes find themselves reused in various contexts. However, it is important to have standard tools that allow analysts to review and assess the outcome of a simulation. OSAMS supports this by standardizing the output data formats that are generated to trace the execution of the simulation. Standard tools can then be developed to support the scientific visualization, statistical analysis, and data mining capabilities that are required to fully analyze the simulated outcome.

4.9 Run Time Performance

Run time performance can be critical, especially in training or course of action assessment and prediction usages. Scalable performance requires flexibility in mapping models to processors and simulations. OSAMS was designed to provide maximum flexibility in this regard. Simulations with tight model couplings can execute on single-processor machines or multiprocessor supercomputers with high-speed communications between the processing nodes. Models with less frequent interactions can operate on different computers connected through a standard network. In any case, OSAMS supports all modes of operation and offers significantly better performance than HLA or DIS alone.

OSAMS also provides the opportunity for simulation engines with different performance capabilities to efficiently host exercises or studies. OSAMS-compliant models could support both real-time and logical time (i.e., run as fast as possible) usages to provide training and analysis. This is in contrast to many current simulation tools that support real-time operation using DIS, but cannot be effectively used to support much-faster-than-real-time analysis.

4.10 Leveraging Investments in Simulation Technology

If OSAMS is developed within a free and open source software implementation, then future science and technology investments developed in this environment will directly benefit all programs using OSAMS. Research institutions such as universities, government laboratories, and industry can all develop innovative capabilities within a free and open source implementation that is shared by all. Instead of R&D projects funded by multiple programs resulting in a technical report and throwaway software, these efforts (if successful) will find their way back into the open source OSAMS code base. To support this efficiently, a common configuration management baseline is required that is not program specific, has robust testing and standards for accepting software changes, and promotes widespread use of OSAMS.

The result will be to focus modeling and simulation science and technology, which will result in better tools, faster run times, cleaner modeling environments, better taxonomy and ontologies for describing models and their behaviors, better strategies for representing human behavior, etc.
4.11 Simulation Tools

OSAMS provides a strategy for developing a common suite of simulation tools. This will lower development costs of supporting multiple tools from different vendors for each simulation system. Operators will only be required to learn one suite of tools that are common for all OSAMS models. Consolidating these tools will lower their overall development cost, while also providing more resources to refine their capabilities. This will have a significant impact on the cost of operating a simulation.

4.12 Open Market Participation

Through standards, OSAMS promises to promote an open market for participation by industry. Various tools, including simulation engines, utility suites, graphical interfaces, and even models can be developed by industry in support of OSAMS. By creating a competitive market through the formation of standards, programs will be able to choose from the best and most cost effective capabilities available. This may result in lower life-cycle maintenance costs for government programs, with a long-term strategy to transfer technology from the government to industry.

OSAMS has the potential to effectively support three standard software business models: (1) open source, (2) GOTS, and (3) COTS. Without such standards, there is no market for these business models to thrive; resulting in the expensive stovepipe integration world that currently exists today.

5 OSAMS Development Strategy

To implement the five essential capabilities described in this paper (composability, abstract interfaces, standard modeling framework, distributed objects, and trace file generation with data logging), OSAMS should evolve in maturity over three natural phases. Note that Phase II and Phase III efforts can be performed in any order, or even simultaneously to support a more aggressive development schedule. Several open source and freely available simulation kernels are available today\(^8\) that support many of the OSAMS capabilities. The necessary refinements and middleware capabilities needed to quickly support OSAMS can greatly leverage existing functionality from these freely available open source tools.

5.1 OSAMS Development: Phase I

The first phase specifies and documents all of the architecture-related interfaces that may be invoked by the models. These interfaces must support the five technical areas briefly described in the previous section and should take into account the capabilities provided by existing simulation engines currently being considered for use within the DoD. If desired, simulation developers can implement the OSAMS interfaces themselves within their own engines. Model dependence on portable utility libraries should be encouraged, but software services relating to the five areas described above must adhere to the OSAMS interfaces without using any other services provided by specific simulation engines. Otherwise, standardized interoperability will be lost. In the future, perhaps OSAMS will also specify commonly used utility services that can be shared by all models.

It is not necessary for each modeling and simulation application or framework to support all of the specified interfaces. Only the interfaces actually used by the models in the specific simulation application must be supported. However, any simulation engine that does not support all of the OSAMS interfaces has limited utility because it may not support interoperability with other models relying on additional OSAMS services. Full model interoperability will be supported as long as at least one accessible simulation engine provides full OSAMS capability. It would be best if such a simulation engine were free and open source. This leads to a second and third phase effort, where the OSAMS is implemented in one or more different simulation systems.

5.2 OSAMS Development: Phase II

The second phase oversees the development of a common middleware software infrastructure with appropriate hooks to allow any simulation engine to implement a full mapping of OSAMS to its internal event-scheduling interfaces. Such a middleware solution will significantly reduce the cost of making a simulation engine OSAMS compliant. It will also help ensure that the interfaces are both implementable and usable. However this phase requires the actual development of the middleware capability. One technical issue will be to define easy-to-use interfaces that minimize the effort of hooking up the middleware implementation of OSAMS to generic simulation engines, especially if the OSAMS interfaces support advanced modeling and simulation constructs. The middleware may need to be implemented in multiple languages such as C++ and Java. Ultimately, the middleware solution should allow any compliant simulation engine to host OSAMS models.

To carry out this phase, it will be important to choose several existing simulation engines as beta testers to

---

\(^8\) Examples of freely available simulation engines that could provide a foundation for OSAMS are: the Missile Defense Agency version of SPEEDES (GOTS), the JSIMS Common Component Simulation Engine extensions made to SPEEDES (GOTS), and the WarpIV Kernel (public/open source).
ensure that the middleware solution supports different engine implementations. By the end of this phase, several existing legacy simulation engines will support the full set of OSAMS interfaces.

5.3 OSAMS Development: Phase III

The third phase supports and encourages the development of OSAMS-compliant simulation engines. Such engines can be built from scratch, provided by extending the capabilities of existing engines, offered as commercial or free open source products, take advantage of the OSAMS middleware, and potentially execute on disperse computing systems such as highly mobile personal computers, networks of personal computers or workstations, and big-iron supercomputers. A freely available open source OSAMS-compliant simulation engine would greatly benefit the broad modeling and simulation community because it would consolidate costs and focus new Science & Technology development. However, intellectual property rights issues may pose problems in terms of life cycle maintenance, software licensing rights, configuration management, industry buy-in, etc. Nevertheless, a long-term goal of OSAMS must be to promote and encourage the development of OSAMS-compliant: (1) component-based models that can be stored in a repository, and (2) simulation engines from industry, government, and academia.

Several freely available simulation engines exist that support most of the OSAMS capabilities. The first engine is the government-owned Common Component Simulation Engine (CCSE) developed by SPAWAR Systems Center for the Joint Simulation System (JSIMS) [27,28,29]. CCSE was based on the Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES) with a number of important extensions that are necessary to support OSAMS. A more scaled-down version of SPEEDES is currently used within the Missile Defense Agency (MDA) to support the Ballistic Missile Defense System Simulation (BMDS SIM). While this version of SPEEDES is being promoted as the basis for future MDA simulation programs, it does not provide many of the important technologies that are necessary for implementing OSAMS.

Another implementation is the industry-owned, yet free and open source, WarpIV Kernel [46] that is currently licensed to more than 60 different organizations, including the United States Government. WarpIV, developed primarily on government-provided Small Business Innovative Research (SBIR) funding, was designed to be backward compatible with both CCSE and the MDA version of SPEEDES. It implements a proposed Open Modeling and Simulation Architecture (OpenMSA) [45] with a cleanly layered next-generation design that spans a wide range of communications and simulation technologies.

These freely available government-funded systems would require additional software modifications to support all of the capabilities necessary to develop a fully operation implementation of OSAMS. However, with moderate extensions, any of these simulation engines could be used today to support highly interoperable overarching model development for the DoD.

6 Summary and Conclusions

This paper began by describing the high cost and technical challenges of developing federation-based simulation systems that connect multiple simulations in a network environment using a common message-passing run time infrastructure such as the High Level Architecture RTI. While the federation approach towards interoperability and reuse has gained momentum within the DoD modeling and simulation community, it is the contention of this paper that a revolutionary, and not evolutionary, approach is needed to significantly lower the cost of simulation and to broaden its usage. As previously stated in the paper,

The problem is not that we need to do things better, we need to do things differently!

This paper then discussed how to provide interoperability and reuse through plug-and-play technologies that would allow operators to directly compose models from reusable model components. Such models could operate on machines ranging from laptops to networks of supercomputers (and everything in between). To reduce the cost of M&S, the primary paradigm shift proposed in this paper is that:

We need to stop building simulations, and start building plug-and-play model components.

Five technical areas were identified as critical focuses for the establishment of the Open System Architecture for Modeling and Simulation (OSAMS). These five technical areas are:

1. Flexible hierarchical composition structure
2. Standard modeling framework
3. Abstract polymorphic methods
4. Distributed object techniques
5. Consolidated trace file generation and data logging

These topics were then discussed at a moderate level of detail to provide background on their functionality. The paper also discussed how integration with legacy simulations through standards such as HLA, DIS, TENA, and future web-based SOA Network Centric Enterprise Systems (NCES) operating on the Global Information
Grid (GIG) could also be automated within a common simulation engine that follows the proposed OSAMS standard. This will significantly lower the cost of next generation modeling and simulation tools in support of emerging programs such as the Future Combat System (FCS) [15].

While this paper addresses the most critical technologies that are required to develop interoperable OSAMS-compliant models, the science and technology community should not ignore other important simulation technologies such as: (1) graphical cataloging and composability tools that incorporate ontology information to determine which models can actually interoperate, (2) human behavior representation and modeling through rule-based state transition schemes, and (3) scalable high performance data exchanges using publish/subscribe enterprise network services that will eventually be provided by the Global Information Grid.

Finally, this paper discussed a number of areas where OSAMS could save significant costs and provide enhanced benefits to modeling and simulation programs. If the costs for developing OSAMS into a widely used standard are shared among programs, then these cost savings will provide significant cost savings to the DoD modeling and simulation community.

7 Acknowledgements

This paper was written by members of the Software Support Activity (SSA) Integration and Test (I&T) team for the Joint Program Executive Office - Chemical and Biological Defense (JPEO-CBD) program [26].

8 Biographies

JEFFREY S. STEINMAN, President & CEO of WarpIV Technologies, Inc., 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 various NASA space exploration missions.

While at JPL/Caltech, Dr. Steinman pioneered the development of the SPEEDES framework. This work resulted in five patents and more than fifty technical papers in high-performance computing, optimistic discrete-event simulation, data structures, message-passing algorithms, object-oriented design, parallel and distributed multi-resolution interest management, parallel high-speed communications, and HLA.

Dr. Steinman has led numerous software development efforts on mainstream simulation programs 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), and the High-Performance Computing Run Time Infrastructure (HPC-RTI). He has also provided consulting and software support for many other programs including the Joint Warfare System (JWARS), the Extended Air Defense Test Bed (EADTB), Enhanced Naval Warfare Gaming System (ENWGS), and several others.

Dr. Steinman was a charter member of the Time Management and Data Distribution Management working groups during the formation of the HLA standard. Dr. Steinman was the lead architect for the JSIMS Common Component Simulation Engine, and now leads the development of the next-generation open source WarpIV Simulation Kernel.

Dr. Steinman is currently supporting various modeling and simulation programs as a member of the Software Support Activity team within the JPEO-CBD program. He is currently organizing the Open Source Initiative for Parallel and Distributed Simulation (OSI-PDMS) study group with SISO.

JENNIFER PARK is an engineer with the Space and Naval Warfare Systems Center San Diego with over eighteen years of C4ISR experience. Currently, she leads JPEO CBD SSA Integration and Test and is responsible for M&S, VV&A, and integration support. She also leads the Navy's M&S VV&A effort and is responsible for developing policy, overseeing its implementation, and promoting its related technology. She graduated from University of Missouri with Bachelors in Electrical Engineering and from the Naval Postgraduate School with a Masters in Electrical Engineering.

BRUCE “WALLY” WALTER is a Senior Program Manager with L-3/Titan Corporation. Wally has been intimately involved in developing and fielding M&S programs for over ten years. He has led military-oriented software development teams for fifteen years. Mr. Walter is currently involved in M&S efforts associated with the Chemical/Biological Defense Program. Mr. Walter has a Master of Science degree in Systems Management from the University of Southern California and a Bachelor of Science degree from the U.S. Naval Academy.

Wally Walter passed away June 1, 2007. He is deeply missed by all who knew and worked with him.

NATHAN DELANE is employed at EG&G Technical Services, Dahlgren VA and is currently a member of the Joint Program Executive Office (JPEO) for Chemical and Biological Defense (CBD) Software Support Activity (SSA). Prior to joining the SSA, Mr. Delane was a
member of the NAVSEA Dahlgren Accreditation Team (NDAT) and spent almost five years working in the area of Verification, Validation and Accreditation (VV&A) of Models and Simulations (M&S). Mr. Delane initially worked for the Information Technology side of NDAT before transitioning over to support various M&S efforts in their accreditation events. He now works as an SSA team member in the area of Integration and Test (I&T) providing direct support to the JPEO-CBD in the areas of M&S and VV&A. Mr. Delane has a Bachelor of Science (BS) in Information Systems from Park University and a Masters in Business Administration (MBA) from University of Mary Washington.

References