A Parallel DEVS Approach for Cloud Simulation Standards

Keith Snively and Dr. Joe McDonnell
Dynamic Animation Systems
12015 Lee Jackson Highway
Fairfax, VA 22033
703-503-0500
ksnively@d-a-s.com, jmcdonne@d-a-s.com

COL Robert Kewley and Dr. Vikram Mittal
U.S Military Academy
Department of Systems Engineering
4th Floor, Mahan Hall
West Point, NY 10996
845-938-2700
Robert.Kewley@usma.edu, Vikram.Mittal@usma.edu

Keywords:
Distributed Simulation, Composability, Reuse

ABSTRACT: Distributed simulation environments present numerous challenges to the modeling and simulation community, including difficulty in creating new models, uniformity of model behavior across a simulation execution, reasoning about execution of models within complex distributed systems, and deploying simulation executions to a cloud environment. Several rapidly evolving technologies can help address many of these challenges. These technologies include functional programming, reactive systems development, microservices, and containerization. In this paper we discuss these technologies and how they can be leveraged within a PDEVS environment. The Parallel Discrete Event System Specification (PDEVS) formalism allows for execution of concurrent complex systems composed of discrete components coupled in a hierarchical structure. We discuss the development of a simulation framework using the above technologies as part of the Distributed Modeling Framework (DMF) project and its use in support of a small arms lethality study. Throughout, we discuss on a set of standards that can enable the definition and composability of models as well as the deployment, discovery, and orchestration of models within a cloud environment.
1. Introduction

Modeling and simulation (M&S) sharing and reuse have always been an elusive goal for Department of Defense (DoD) M&S community [1]. Some of the barriers to reuse include discovering the M&S resource, assessing the suitability of the resource for reuse in its new environment, interoperability between resources, and the potential for misuse [2]. Despite these challenges, there are potential cost savings and decision making benefits that could be realized with proper reuse. These benefits include better understanding of complex systems by allowing analysis of the parts, simplified testing of systems, M&S cost control, and improved M&S maintainability and usability [3]. One approach to improving simulation composability and reuse is to look to the software engineering community. In so doing, we must still account for the unique characteristics for simulation models [4]. Within the software engineering community, some significant advancements offer an opportunity for the M&S community to borrow from their innovations and rethink the notion and cost vs. value proposition of M&S reuse.

The current DoD model of deploying M&S as software applications inhibits reuse. Recent advances in functional programming, reactive systems, cloud computing, containerization, and the associated growth of microservice architectures offer an opportunity to leverage the cloud for deployment and widespread use of M&S services. A Defense Modeling and Simulation Coordination Office (DMSCO) study outlined the benefits and barriers for cloud-based M&S and recommended investment in areas such as service oriented architectures (SOA), composability, and new M&S architectures appropriate for the cloud environment. The report identified additional benefits to include on-demand and accessible M&S resources, rapid elasticity to support dynamic needs for intensive computation, resource pooling to reduce IT requirements, M&S available for users to purchase on a pay-per-use model, and reduced cost of simulation-based exercises [5].

In order to realize these advantages across the DoD M&S community, we must do more than simply deploy models and services to the cloud. Individual components must be designed and built with composability in mind. This requires a series of standards to describe and discover services, to describe discrete event models, to compose those models, and to automatically deploy and execute them in the cloud. A joint research effort between the Army Research Lab Human Research and Engineering Division Advanced Training and Simulation Division and the West Point Department of Systems Engineering has begun to illuminate techniques to realize some of these goals. From this research, we introduce requirements for standards and discuss areas where additional research is needed before effective standards can be proposed.

2. Supporting Technologies

Before discussing appropriate standards to realize cloud-based deployment and widespread reuse of models and simulations, it is important to identify key software engineering and deployment technologies leveraged for this purpose and how they relate to goals of the M&S domain. In section 3, we discuss a use case of these technologies for supporting a constructive simulation.

2.1 Functional Programming

A couple of characteristics of pure functions are particularly useful to enable M&S reuse. First, a pure function has no side effects. It does not read or write data or change internal variables. Second, the output of a function is uniquely and consistently determined by its inputs. Under these conditions, functions are composable. A function can be passed as an input argument to another function with no impact of the result of the computation. This property is known as Referential Transparency. This property is important as it makes it easier to reason about the behavior of a program and reduces the potential for misuse. For our purposes, a pure function is the smallest element of simulation reuse.

2.2 Reactive Systems

Modern computing systems must respond to increased demands for responsiveness and availability even in the face of ever increasing scales with respect to processing power and data. These demands cannot be met by simply adding storage and processing to a single computer – scaling up. It requires the coordinated use of multiple computers, typically deployed in cloud environments – scaling out. The Reactive Manifesto [6] describes a set of design principles for the development of reactive systems: They must be responsive to user input in a consistent and timely manner. They must be resilient in the face failure, isolating and handling failure in such a way that the entire system does not crash. They must be elastic, responding to increased demand with increased resources. And finally, they must be message driven with asynchronous messages passing data and coordinating system events in a non-blocking fashion.
2.3 Microservices

A growing trend in the development and deployment of services and applications is Microservice Architecture. This design paradigm breaks a service into a group of loosely coupled microservices, each running in its own process space, and communicating over lightweight protocols. While there is no formal definition of a microservice, proponents assert they share many characteristics [16]. These include:

- **Single Responsibility**: A microservice fulfills a single purpose. For instance, a ballistic model runs as a separate service from a weather model.
- **Separated Code Base**: The microservice uses its own code base. This allows each service to evolve more independently from other services.
- **Self-Contained Process**: The microservice operates in its own process space, and failures do not directly affect other processes. This also means the microservice is implemented in the programming language that best supports its function and is not tied to the implementation language of other microservices.
- **Database Independence**: The microservice accesses its own database. The database server may or may not be shared with other services, but the database itself is not shared. This allows changes to the database, as well as choice of type of database and server, without affecting other services.
- **Lightweight Protocols**: A microservice communicates through a lightweight, and language agnostic, protocol. The lightweight protocol simplifies development of request-response interaction. Generally, choreographed communication, such as provided by AMQP, is preferable to orchestrated communication, such as creating point to point connections.
- **Individual Deployment**: A microservice can be deployed independently, without redeploying an entire application. This enables fielding new features rapidly and addressing defects that only affect one piece, or a small portion, of an application.

Microservice Architecture can be seen as a form of SOA, or even an example of SOA done right [20]. Whether or not this is true largely depends on the definition used for SOA and microservices. The important point is many of the benefits of SOA are shared with microservices, while microservices provide a more concrete approach.

There are several drawbacks to microservices. One often cited is increased complexity in deploying applications. Containerization, discussed in the next section, has gone a long way to simplifying this process however, and in fact, the rise of microservices can be tied to the rise of containerization technology. Another is that microservice architecture does not completely remove the impact of changes of one service affecting another. For example, changes to the network application programming interface (API) can require changes to other services utilizing that interface. The Tolerant Reader [22] and Consumer-Driven Contract design [23] design patterns can help mitigate the impact of changes to an extent. These patterns allow for a degree of backward and forward compatibility with the message formats between the networked services. This is also an area where standardization can help manage the cohesion between services.

Another issue is the latency and performance overhead of fielding a service. Having microservices become too fine-grained can become problematic, and is sometimes dubbed a nanoservice, and considered an anti-pattern [21]. For instance, having coordinate conversion as a service entails too much overhead for such a frequent function to be practical and is better served as a library. Determining the proper granularity for models as microservices is part of the Distributed Modeling Framework (DMF) project.

Still, microservices have numerous benefits. The decentralized nature of microservices fits well with the organization of the DoD within branches and functional areas. A given organization within the DoD will fill a specific function and retain subject matter expertise within that area. These organizations develop models for their domain for use within the modeling and simulation community. Currently, these models are often implemented by third party organization within a proprietary or monolithic simulation application. Getting access to these implementations can be difficult, requiring distribution agreements with one or more organizations. Getting access to just the isolated model implementation can be impossible as it is integrated into a monolithic application. A microservice approach, on the other hand, allows the developers of the model to also provide the implementation and continuously deliver or deploy the model. Continuous Delivery is a development methodology that uses rapid development cycles and ability to provide release of any version with a simple, if not fully automated, process. It builds upon Continuous Integration, where developers frequently commit changes that are automatically tested against the baseline with results provided back to the developers. Continuous Delivery adds additional testing in deployment environments, so software can be reliably delivered at any time. Continuous Deployment takes this process a step further and provides automatic deployment of new releases. These processes allow users of a model ready access to any version of the model as desired.
2.4 Containerization

Another growing trend is deployment of applications within containers. Containers allow wrapping of an application, along with all necessary dependencies and system libraries to run the application. This allows running the application with minimal requirements of the host system. Containers share kernel space with the host system, so do not require hardware virtualization and can run at near bare metal speeds [24]. Docker has popularized containers and added tools to easily create, store, and deploy containers. Docker adds a registry, which can be public or private, for storing container images. The docker executable is able to pull an image from the registry and locally deploy it as a running container. Users can modify and commit a container to the registry. The images use a layered file system, so only the differences need to pushed to the registry, which are stored as a layer built atop existing images [25]. These features make Docker containers particularly well suited to microservice deployment.

Kubernetes allows deployment of containers in a cloud environment. It is an open source system that currently works with Docker containers. Kubernetes deploys containers, which it terms pods, across one or more servers, called nodes, and manages the lifetime of the pods. It actively monitors health of the pods and restarts them if they fail. Kubernetes also allows the definition of services, which serve as static endpoints for accessing pods, regardless of where in the node cluster the pods are running. Pods within a cluster reference services by name using an available cluster DNS service, aiding in the orchestration of services (or microservices). The service endpoints perform round robin load balancing to a set of replicated pods, which provide the service implementation. This allows scaling out of services by simply adding nodes to a cluster. Deployments of pods and services within kubernetes are described using yaml files or json notation.

An advantage of Kubernetes to other cloud deployment technology is portability. It is the engine behind the Google Cloud Platform and it also runs under AWS EC2, Rackspace, Microsoft Azure, a private cloud environment or on bare metal [26]. This provides flexibility to the user to determine the best environment for supporting a simulation environment. It also allows for reusing much of the same containers and deployment configurations within cloud and private environments. This portability supports DoD users who may or may not have access to cloud services during a particular simulation event.

3. Weapons Lethality Service - An example

The Weapons Lethality Service (WLS) is a simulation put together to support a study of small armament effectiveness, leveraging the above technologies. The WLS uses 5 base services:
- Acquire Service: Provides target acquisition and identification performance.
- Small Arms Accuracy Service: Provides weapon accuracy performance.
- Rate of Fire Service: Provides fire and burst rate.
- Vulnerability Service: Provides damage assessment.
- Human Body Service: Provides a human body representation used to assess target impact location.

The first four services are implementations of the Army Materiel Systems Analysis Activity (AMSAA) physical knowledge acquisition document (PKAD) for those models. Each service was independently verified and validated against the PKAD.

The WLS employed implementations of the technologies and potential standards described in this paper. It also revealed areas where standards are lacking. The base services are implemented using a functional programming in a reactive system. The implementations are written in Scala and behavior encapsulated using Akka actors. The transparency of actor location allows the models to be run with the same process as one another or broken out in remote services. These stateless actors can easily be placed in containers or run as a microservice. For the purposes of experimentation, the Acquire model is run as a microservice, which can be contacted through Akka messaging or through an HTTP interface.

The Weapons Lethality Service implements the simulation logic within a PDEVS framework consisting of several components, each of which is itself a PDEVS model. This DEVS Distributed Modeling Framework is further discussed in section 5. The Soldier coupled model consists of two atomic models. The sensor model uses the Acquire service and the rifle model uses the Small Arms Accuracy and Rate of Fire services. The Target atomic model uses the Human Body and Vulnerability services. A higher level root coordinator model contains these as PDEVS models and manages the overall simulation messaging and simulation time. The PDEVS models store the state variables during simulation execution and call the underlying services, which are stateless, as necessary.

The WLS was deployed as a standalone application and as a web application. Ultimately, in the latter case, the WLS was deployed as a container into a private Kubernetes cluster. The Simple Build Tool (SBT) Native Packager provides the ability to create a docker container image for an application over a user customizable base image. The base image container must contain a Java Runtime Environment and support executing bash scripts. These images can be published locally to
docker or pushed to a remote registry. We created a private docker registry for holding packaged applications and connected it to the Kubernetes cluster. This capability enabled Continuous Deployment of WLS and supporting models as changes were committed to the code repository.

The fast execution time of the WLS allows it to be used as a screening model for an Army weapons study. It executed a large scale design of experiments with 7 decision variables and 11 environmental variables to assess the performance of different weapons configurations. Analysis of the performance of these weapons will allow the Army to make an informed decision about which configurations to analyze in a much more complex and slower running combat simulation model. In this particular study, the microservices approach allowed the development team to efficiently build a small and fast simulation that focused on the key models affecting the decision.

4. Standards for Composable Services

The best way to ensure composability of services is to represent them as pure functions with no side effects. This guarantees that service or simulation can call these services and depend on the result. It also ensures that the service does not manage state. One of the biggest challenges for simulation composability is managing state across disparate systems. Architectures such as High Level Architecture accomplish this, but these architectures are very complicated, and even simulations that comply with HLA have differences in implementation which inhibit interoperability. However, a stateless service can be used by any computing architecture.

We discuss in Section 5 a method to manage state within models. The important part for services in stateful systems is to represent the service as a state transition function. If the purpose of a service is to calculate state, it must take as inputs the current state of the system and the time elapsed since the last state transition. It will return the subsequent state of the system as an output. Services must also handle random numbers in a stateless way. If a service requires a random number, it should take that number as an input and not use its own internal random number generator. If it did, that would be a side effect, and the service could return different outputs for the same input.

Services should be developed in a way that maximizes the composability options of potential users. Specifically, a user should have the option to call the service in its native language, exposed as an API, because this can increase cohesion in application design and reduce unnecessary coupling. Of course, many potential users will be working in different languages. In this case, the service should present a network interface that allows it to be called remotely from distributed systems. If we are following the principles of reactive systems [6], then this should be an asynchronous messaging interface. For example, the simple service shown in Figure 1 illustrates an API with input and output data requirements. The developer can use this API directly in its native language, or he or she can call the service using Protocol Buffer messages using the networked endpoint. Finally, since functions can be composed, services may call other services during execution as long as those services are also implemented as pure functions.
Finally, the service should be discoverable, understandable, and usable by those that need to use it. These are important and distinct features, so we will address each one separately.

- **Discoverable:** The first type of discoverability applies to simulation developers who are looking for well-developed services to integrate into their architectures. They must be able to find these services in a known repository and pull metadata that explains how these services work.

- **Understandable:** According to the Levels of Conceptual Interoperability Model [7], net-centric data standards and service description standards such as the Web Services Description Language (WSDL) [8] only achieve level 2, aligned static data. Full level 4, conceptual interoperability, may be obtained by further describing the dynamics and semantics of services using an approach similar to that of the Base Object Model (BOM) [9].

- **Usable:** Users need the ability to pull, run, and access the services themselves either via an API or via a networked service endpoint. In a cloud and containers environment, hosting the services in a container repository and managing deployment endpoints using one of the cloud-based service discovery technologies such as etcd [10] will allow distribution, execution, and connections to these services.

To summarize, many different types of simulation architectures can make use of services to perform calculations or state transitions if those services are represented in a standard way, as pure functions, with no side effects. The simulation system manages state, not the function. Standards for these services that address data, interfaces, dynamics, and semantics will improve composability. Finally, cloud deployments necessitate the development of standards for accessing services via a container registry and service discovery and coordination mechanism.

5. **Standards for Composable DEVS Models**

One solution for distributed state management is to develop simulations as compositions of DEVS models [11]. The authors are part of a research team that has developed the DEVS Distributed Modeling Framework (DEVS DMF) [12] which can be accessed via Github [13]. This system adapts the Parallel DEVS algorithm for execution in a distributed asynchronous messaging environment using the actor model of computation [13]. Figure 2 shows an atomic DEVS model implemented as a SimActor. Simulation services implemented as described in Section 4 can be used as state transition functions for this model. If services are the most granular element of reuse and composition, a DEVS model is the next.
DEVS models manage state for a simulation system, and the SimActor encapsulates that state so that it can only be changed via messages received. Several elements work together within the SimActor to perform this role. The state manager maintains a set of data structures representing the dynamic internal state of the entity modeled by the DEVS model. Static properties are those properties of the modeled entity initialized at the start of the simulation but do not change during execution. The schedule, also a component of internal state, keeps track of future events. Finally, via the Split Stream Actor, each SimActor receives a distinct segment of the random number stream to be used to generate random numbers to be passed as inputs to the state transition functions.

Figure 3 shows the elements of a complete DEVS DMF simulation. A Model Simulator executes a DEVS model over time using the Parallel DEVS algorithm. A Model Coordinator coordinates the actions of multiple internal DEVS models by routing messages to and from their Model Simulator. It represents a coupled DEVS model. The Root Coordinator controls execution of the entire simulation while the Split Stream Actor manages random numbers and the SimLogger logs simulation data and events. Looking at Figure 3, one can see that a DEVS simulation is hierarchically composed of atomic DEVS models and coupled DEVS models. The elements of composition in this framework are, in order of granularity, the simulation service, an atomic DEVS model, and a coupled DEVS model. Any of these components can be pulled and reused in other simulations.
The DEVS DMF framework is an implementation of what could be a standard. It operates via a series of messages that are passed between the Root Coordinator, Model Coordinators, Model Simulators, and DEVS models. A standard for DEVS DMF should allow other implementations in other programming languages and coordination and interoperability between these different implementations. Therefore, the standard must begin with a serializable data specification of the coordination messages of the DEVS DMF framework and a clear specification of how to specify the structure of event messages passed between models. Candidates for this specification include Protocol Buffers, XML, JSON, and BSON. The second part of this standard would describe the expected behavior of components as they receive messages from other components.

Just as reuse and composition of services require them to be discoverable, understandable, and usable, the standard for composable DEVS models must similarly achieve this through approaches similar to WSDL, BOM, cloud container repositories, and service discovery technologies.

6. Standards for Messaging

The current version of DEVS DMF is implemented in Scala using the Akka framework [13]. Akka implements location transparency of actors so that different elements of the architecture can be deployed in different cloud based containers. However, if the standards are going to support communication between different implementations, outside the Akka framework, then a standard for asynchronous communication via messaging must be defined. In this architecture, each service or a DEVS component will need to define, in addition to its component endpoint, one or more messaging interfaces to which components will send messages to be forwarded to the simulation component. This specification will need to include message data structure as well as protocol definition. A standardized format for capturing this information could be beneficial, especially for code generation processes. Such a standard could also encourage use of patterns, such as Tolerant Reader, which helps reduce impact of extensions to message structures on existing services [22].

7. Standards for Cloud Deployment

Deploying simulation services to the cloud typically requires knowledge of configuration points for each service. This configuration information includes communication ports, environment variables, database connections, external service connections, file system volumes for input and output, and live-ness probes. This information is required whether deploying as a container, Virtual Machine, or some other service. For containers, this information is typically given on a description page, such as on dockerhub, in human readable form with several deployment examples. This can complicate deployment of simulations composed of a large number of services, as users must manually configure each. Further, this documentation typically covers basic configuration and requires referencing documentation on the encapsulated service for deeper configuration points. Standards to describe this type of meta-data can aid in locating and deploying services in an automated fashion, especially where complex orchestration or choreography is required. Such standards could apply to more than just containerized services and applications. This meta-data could be stored with the container (or deployable application) and/or in an independently searchable registry.

Standards can also help capture the actual deployment configuration for a given execution environment. This could take the form of an extension to the SISO Federation Engineering Agreements Template (FEAT), which includes information on the design and management of a federation [17]. This could be used to automatically generate deployment scripts and files for a particular cloud environment, such as yaml files used by Kubernetes.

Cloud deployment further entails unique concerns when creating simulation environment. These concerns can include performance, latency, and reliability. It also requires weighing capabilities and costs of differing public and private cloud environments. These issues will impact the engineering process while designing simulations. A candidate for such a process standard could serve as an overlay to the existing IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP), which covers engineering process in creating simulation executions [18].

8. Development and Tools

The microservices approach presents several challenges with respect to development complexity, data collection, and debugging. A distributed discrete event simulation system adds an additional factor, virtual time, to the challenges. In order to develop, integrate, and analyze simulation systems using this framework, a set of tools is necessary. This research program intends to look into the viability of the following supporting tools.

- A standards-based graphical development environment that allows developers to compose state transition functions, atomic DEVS models, and coupled DEVS models. The interface could be tied to a domain specific simulation language and supporting code generation system that significantly reduces the boilerplate code needed to instantiate and integrate models.
• A standards-based monitoring and debugging capability. During the verification and debugging phase of simulation development, the user would be able to collect a high degree of logged messages, state transitions, and state variables from the simulation system – all correlated with virtual time stamps and real time stamps. This data could then be automatically composed into traces that support model verification and debugging.

• A standards-based data collection and analysis capability. During distributed simulation execution, analysts would be able to establish tags and conditions for the logging of certain messages, state transitions, and state variables to support data collection and analysis of simulation results.

9. The Way Forward

A standardization effort requires commitment and funding from a community of interest working on common problems. This paper discusses some areas where we believe standards can be beneficial as a starting point to identify these common areas. A single standard need not attempt to address all areas discussed in this paper, and indeed, some areas are currently more well-defined, such as PDEVs modeling, than others, such as cloud deployment. A specific course, such as a Study Group, will be determined by level of interest by the community and those that fund it.

However, the approaches described in this paper, along with other organizations’ related research, provide a significant opportunity to introduce a new set of standards to support M&S interoperability and reuse – based on patterns from modern software engineering and enterprise integration. Sticking to widely used approaches yields the best chance of having standards that can be realistically implemented and are supported by open source and commercial tools. In addition, the M&S community must augment the existing tools with specific capabilities to support debugging, verification, and analysis of simulations.

10. References

[18] IEEE Std. 1730™-2010, “IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP)”


**Author Biographies**

**COLONEL ROBERT H. KEWLEY** is currently the head of the West Point Department of Systems Engineering. He is a career Army officer with a background in Armor. He is a graduate of the West Point Class of 1988 with a degree in mathematics. He holds a Master’s of Industrial Engineering and a Ph.D. in Decision Science and Engineering Systems, both from Rensselaer Polytechnic Institute. Colonel Kewley’s research interests focus on simulation and command and control systems.

**JOSEPH S. MCDONNELL, Ph.D.** is a Principal Scientist and Director of Modeling and Simulation with Dynamic Animation Systems, Inc. He has over 20 years of experience with modeling and simulation supporting various United States (US) Government programs working in technical, staff and managerial positions. Currently, he supports Science and Technology programs at the US Army Research Laboratory, Human Research and Engineering Directorate, Simulation and Training Technology Center as a Principal Scientist. Dr. McDonnell received his Doctorate of Philosophy (Ph.D.) in Mathematics from the University of Virginia.

**VIKRAM MITTAL, Ph.D.** is currently an Assistant Professor in the West Point Department of Systems Engineering. He holds a BS in Aeronautics from Caltech, a MSc in Aerospace from Oxford, and a PhD in Mechanical Engineering from MIT. Dr. Mittal’s research interests focus on M&S, robotics, and energy.

**KEITH SNIVELY** is currently a Principal Software Engineer for Dynamic Animation Systems. He has nearly 20 years of experience in developing distributed simulations and simulation protocols for the Department of Defense. Mr. Snively holds a Master’s of Science degree in Mathematics from the University of Virginia.