DEVS Distributed Modeling Framework - A Parallel DEVS Implementation via Microservices

Robert Kewley
Dept of Systems Engineering
U.S. Military Academy
robert.kewley@usma.edu

Neil Kester
Army Materiel Systems
Analysis Activity
neil.e.kester.mil@mail.mil

Joseph McDonnell
Dynamic Animaton Systems
joe.mcdonnell@d-a-s.com

ABSTRACT

This paper introduces DEVS Distributed Modeling Framework (DEVS-DMF), a publicly available implementation of DEVS for integrating simulation models as parallel and distributed microservices suitable for containerization and automated deployment. Based on the Parallel DEVS framework, DEVS-DMF explicitly enables asynchronous execution of distributed models via the actor model of computation. It is optimized for interoperability of models and flexibility of deployment, as opposed to performance. The DEVS-DMF framework provides two simplifying benefits to the simulation developer. It encapsulates internal state and synchronizes simulation execution to free the developer to focus on designing the flow of messages and encoding the state transitions and outputs generated by their flow. It also provides a natural model for partitioning the simulation into local processes, because each DEVS model, as an actor, performs a unit of execution on a separate thread each time it handles a message. An additional advantage of this framework is location transparency, its flexibility to deploy actors in a single Java Virtual Machine to scale up, or to deploy actors to different containers in the cloud to scale out. Two simulation implementations of DEVS-DMF include a parallel simulation test case and a combat weapons simulator that assesses the performance of alternative small arms weapons designs.

Author Keywords

DEVS; Parallel Discrete Event Simulation; Distributed Systems; Cloud Computing

ACM Classification Keywords

D.1.1: Applicative (Functional) Programming; D.1.3: Concurrent Programming; D.2.12: Interoperability; F.1.1: Models of Computation; I.6.1: Simulation Theory;

1. INTRODUCTION

Distributed and parallel discrete event simulation in the cloud is an emerging research area driven by the cost advantages of scaling simulations out with cheaply available on-demand computing resources. Cloud and container technologies are also driving development of software from large monolithic systems designed to run on a single hardware platform to distributed microservices that can be hosted on a variety of platforms and distributed over a network[16].

The many challenges of deploying parallel and distributed simulations to the cloud are discussed by Guan[7], Yoginath and Perumulla[22], D'Angelo and Marzolla[4], and Fujimoto et. al.[6]. In summary, these researchers seek features such as efficient scheduling and management of execution, separation of concerns so that simulation developers do not also have to manage parallel and distributed scheduling and execution, a natural model for partitioning the simulation, usability across a variety of cloud providers, flexibility in scaling both up (better hardware) and out (multiple machines), the ability to use existing distributed services, management of deployment and execution in the cloud, management of cost, management of power consumption, and security. These authors have tried and tested several approaches noting some significant challenges with implementation in the cloud.

This paper offers DEVS-DMF as a way to address some of these challenges by combining a parallel DEVS[3] execution environment with the actor model of computation[8]. It generalizes an approach applied by Beraldi et. al. to a microservice architecture[19]. Parallel DEVS manages scheduling via its model coordinators and simulators. It achieves separation of concerns and partitioning by isolating state, event management, and output within DEVS atomic models. Its implementation as Akka[20] actors enables location transparency, which allows flexible up or out scaling within a single machine, to other containers in the cloud, or across cloud infrastructures. Its asynchronous message-based framework provides a pathway for integrating existing microservices, and its pluggable and configurable transport layer provides options for security by taking advantage of security models in the underlying transport layer. It does not currently offer scheduling, deployment management or cost and power management within any specific cloud infrastructure.

Note that DEVS-DMF is optimized for interoperability and flexibility, as opposed to performance and parallelism within high performance domains. While it does inherit the features of parallelism and scalability, its intended use case it to support efficient integration and execution of disparate models that already exist within an organization.

2. DEVS DISTRIBUTED MODELING FRAMEWORK

The DEVS-DMF development team is happy to announce the availability of the first release of the DEVS-DMF infrastruc-

SpringSim-TMS/DEVS 2016, April 3-6, 2016, Pasadena, CA, USA

ture for use in your organization. This framework is appropriate for the integration of existing simulation models and services using enterprise integration technologies[12].

The EASE-DMF program has evolved over the last two years. Our SpringSim 2014 work in progress paper described the introductory ideas and high level requirements, which are repeated here[13].

- The system should allow easy discovery, understanding, and integration of existing computational models.
- The system should support coupling of models with entities, functions, or behaviors in the systems architecture.
- The system should expose input data so that it can easily be manipulated in the development of complex scenarios.
- The system should expose the trajectory of state data so that it can be used in output analysis.
- The system should enable efficient design and execution of experiments.
- The system should allow parallel computation for execution of large-scale experiments.

The current release meets these high-level requirements, and we are currently executing phase II of the research plan. Our 2014 Fall Simulation Interoperability Workshop paper described the advantages of the DEVS-DMF modular approach as compared to High Level Architecture[10] or Distributed Interactive Simulation[9] approaches. The fundamental difference is that previous approaches seek to compose entire simulations whereas DEVS-DMF seeks to break simulations apart into stateless microservices and discrete DEVS models. It composes these more modular elements instead. It also introduced a concept for simulation development from a SysM[18] architecture and showed a proof of principle implementation for an unmanned ground system[14].

Our SpringSim 2015 paper detailed the initial parallel and distributed implementation of DEVS-DMF[13]. It described an overarching enterprise architecture for enterprise modeling and simulation and a set of technology choices for implementation in this research. It also presented DEVS-DMF in the context of design of experiments in support of resilient system design[15]. Despite some initial success, the optimistic time warp implementation based on the work of Nutaro[17] has been fully replaced in the current release by a Parallel DEVS implementation based on the work of Chow and Ziegler[3] due to the advantages presented at SpringSim 2015 by Ziegler[23].

Subsequent development of microservices to support weapons effective analysis and integration of those services by the current release of DEVS-DMF have yielded improvements in robustness, distributed execution, and state management. The remainder of this paper describes the technical implementation, demonstrates results from a parallel simulation test implementation and from weapons lethality analysis, and describes directions for continued development.

3. DEVS-DMF ARCHITECTURE

DEVS-DMF is implemented using the actor model of computation[8][2]. An actor is a computational agent that responds to each incoming message by changing its internal state, or behavior, and sending messages to other actors. Actors can also create other actors in a hierarchical fashion. These features make the actor model a natural way to implement DEVS models. The encapsulation of state as private, only internal to actors, and the message-based communication system lead to efficient handling of multi-threaded and distributed simulations.

DEVS-DMF is built on the Akka framework, an element of Typesafes Reactive Platform that is designed to provide an implementation of the actor model, fault tolerance, location transparency, and persistence[20]. Actor systems communicate exclusively by sending immutable messages to the mailboxes of other actors. Akka naturally supports fast multi-threaded parallel computing and is the foundation of the Apache Spark big data computing engine[21]. Actors run on the Java Virtual Machine in either Java or Scala.

3.1 Top-Level Components

A DEVS-DMF simulation is managed by a series of top-level components to initialize the simulation, manage random numbers, log simulation events, and manage the simulation clock.

Simulation The simulation is the main executable for the series of simulation runs. It reads initial files, starts the actor system, starts the simulation logger, and manages random number seeds for each iteration. The library supports simulation experiments with multiple design points, each having a number of iterations.

Split Stream Actor A DEVS-DMF simulation must coordinate random number generation across a network of computers and processors. This is made possible by a parallel random number generator developed by Kaminsy[11] in his Parallel Java Library[1]. This algorithm splits the parallel random number stream into successive sub-streams of equal size. As each new DEVS model is created, it sends a message to the Split Stream Actor which assigns it the next sub-stream. This ensures each DEVS model uses a different sub-stream in the larger random number stream. The current implementation has a sufficient period to allow one billion different streams of one billion numbers each. Applications which need larger sub-streams or a larger number of streams, can adjust the sub-stream size as needed.

SimLogger This actor receives and logs messages containing data sent by DEVS models in the simulation. It logs each output message sent, each internal event received, and the value of each state variable as it changes over time. It can also log messages containing data from the implementations of state transition functions. It also serves as a placeholder for the planned introduction more sophisticated implementations that write simulation events and state transitions to a persistent data store.

Root Coordinator This root coordinator is adapted from the one described by Chow and Ziegler[3]. It initializes the

split stream actor, starts the simulation, and manages time advance by sending messages to the top-level model coordinator. It also stops the simulation once termination criteria have been met.

3.2 Coordinating Simulation Execution

A DEVS simulation consists of a hierarchy of atomic and coupled models. The parallel DEVS algorithm uses a model simulator to pass execution messages to an atomic DEVS model, whereas a model coordinator replicates the behavior of an atomic model but coordinates the passing of messages to several subordinate atomic or coupled models.

Model Coordinator The coordinator is a DEVS coupled model that coordinates the actions of one or more internal DEVS atomic or coupled models. As an actor, it can exist on the same or different computers than its parent and subordinate models. In its coordination of the actions of internal models, the coordinator maintains several lists which enable asynchronous and distributed execution of its subordinate models. These include a list of each subordinate's next state transition, a list of imminent models, a list of influenced models, which is a combination of imminent models and models to which an external event has been sent, and several other lists simply to keep track of subordinates from which a message is expected. The coordinator executes its models in two phases.

In the generate output phase, the coordinator sends a generate output to each imminent subordinate model. As subordinates produce output messages, the coordinator acts as a message router, receiving the message, performing any necessary mappings, and passing that message either to another internal model as an event or upwards to its parent coordinator. The developer of a DEVS-DMF simulation must write the code to handle the different types of outputs and events managed by each coordinator.

Upon completion of output generation and message dissemination and bagging, the coordinator will get a message to execute state transition. It will combine the list of imminent models with a list of models to which an external event has been sent to create a list of influenced models. It sends a message to each influenced model to execute state transition. At that point, it will await a message from each influenced model notifying the coordinator that state transition is complete and reporting its next scheduled internal transition.

Model Simulator Each model simulator contains an internal DEVS model that it executes over time in response to messages from the model coordinator. It is implemented as an actor. The only internal state maintained by the simulator itself is the list of received external event messages. The DEVS model has the responsibility for maintaining the state of the model.

As simulation execution begins, the simulator first responds to a generate output message by calling the model's output function. The propagation of output messages and generated event messages causes the simulator to receive incoming event messages, add them to its message list, and log receipt of the messages to the SimLogger.

If the DEVS model is imminent, the model simulator will receive an execute transition message, which will cause it to execute the model's internal state transition, external state transition, or confluent state transition, depending on the presence of external events on the event list or internal events on the event schedule.

3.3 DEVS Model

The unit of execution for a DEVS simulation is the atomic DEVS model. In DEVS-DMF, each DEVS model is an internal class of a containing model simulator that coordinates its execution based on messages from a parent coordinator. The internal state of a DEVS model is comprised of four different types of data.

Static Properties These are set at simulation intialization and do not change over the course of a simulation run.

Random Properties These are randomly initialized using the model simulator's parallel random number generator and do not change over the course of a simulation run.

State Variables These are set with an initial value but change over the course of a simulation run based on the results of internal and external state transition functions. Their values are managed by a corresponding SimEntityState object.

Schedule This is a time-ordered list of internal events that will be processed during internal state transitions.

The DEVS model executes internal, external, and confluent state transition functions based on received event data. The default behavior of the confluent state transition function is to first execute the internal state transition function before responding to external events. This can be overridden in implementations where the concurrent arrival of external and internal events yields different behavior.

All of the code to handle time advance and coordinated execution of DEVS models is handled by actors in the library or by code generation. This separation of concerns frees the developer of an individual DEVS model to write the event handlers which are called each time the model executes a state transition in reaction to each type of internal or external event. The model simulator and DEVS model are automatically code generated from the structure of the DEVS simulation, but the event handlers are contained in a separate trait file for which only method stubs for each event handler are generated. In these event handlers, a the model may be influenced in one of three ways.

- Change the value of a state variable
- Schedule an internal event
- Schedule a model output

In addition, an event handler can call an external microservice to compute these changes. The recommended way to do this is asynchronously via the creation of transient subordinate actors to handle the messaging to these services. Three utility objects support management of internal state data.

State Manager The state manager provides access to dynamic state variables and tracks their transitions over time. There are two classes of state variables, those that are known at initialization and have a single value, and those that become known only after receipt of messages during execution.

To understand the difference, consider a security camera that automatically slews from left to right across a scanned area. An example of a known state variable is the direction the camera is currently pointing. An example of an unknown state variable is the location of objects seen by the camera. It will not know of the existence of these objects until they are seen during simulation execution, after which the camera maintains a list of detected objects and their locations.

SimEntityState Each known state variable has a unique name along with a SimEntityState object. For unknown variables, the state manager maintains a list of SimEntityState objects, mapping each to a unique identifier. Every time a state variable changes during a state transition function, this object records the new value, and the simulation time at which the change took place. These changes, to include the most recent change and time, are available as the DEVS model executes state transitions. Note that the state transition functions of an atomic DEVS model take not only the state values, but also the length of time that the entity has been in that particular state[3]. These values are also used for logging the trajectory of state variables during a simulation run.

Schedule Each DEVS model has an internal schedule that is used to schedule future or concurrent internal events. During a simulation run, a DEVS model, through its model simulator, reports to the model coordinator the time of the next internal event on the schedule - this is the time that the model will become imminent. When the internal state transition function is called, the internal event is removed from the schedule and passed to an event handler, written by the model developer, to execute the state transition.

4. ACTOR COMMUNICATION

The DEVS-DMF system coordinates the execution of distributed actors via asynchronous messages. This architecture enables two types of scaling flexibility. It is possible to scale up by deploying actors within a single Java Virtual Machine and by adding processors. In this approach, messages between actors have very little overhead, and communication is extremely fast. However, scaling up by improving hardware can be very expensive. Alternatively, an actor system can be scaled out by distributing actors across multiple machines. Scaling out introduces message passing overhead and a requirement to serialize all messages, but it naturally fits a cloud deployment environment in which more processors are affordably available to simulation users. This flexibility enables a deployment strategy in which models that have a high degree of dependency and resulting communication are deployed within a single Java Virtual Machine - scaling up. Models that are relatively independent or have less communications between them are deployed on different machines - scaling out. Furthermore, in a microservices environment, the services needed by the simulation's state transition functions will likely already be distributed in a cloud. Actor coordination occurs in three phases.

4.1 Simulation Initialization

Messages 1-14 in Figure 1 represent simulation initialization. At the start of the simulation, model coordinators send messages down the hierarchy to request the next scheduled transition of subordinate models. When a DEVS model receives this first request, it sends a message to the SplitStreamActor to get the random seed value corresponding to its section of the stream. It will then initialize any random properties of the model before responding up the hierarchy with the time of its next scheduled transition.

4.2 Simulation Output

Messages 15-30 in Figure 1 represent simulation output. In a DEVS model, simulation output immediately precedes internal state transitions. The model coordinator sends a GenerateOutput message down the hierarchy. Upon receipt of this message by an imminent ModelSimulator, it will call the output function of its internal DEVSModel. This generates an output message to its parent coordinator, which will handle the message by routing either up the chain as an output or back down to other internal models as an external event. In Figure 1, message 20 is an output from the DEVS model that is handled by the model coordinator and sent down to an internal model coordinator as an event message, message 21. That message is bagged to the subordinate coordinator's external event list, and message 22 is sent confirming receipt and bagging. When the model coordinator receives all BagEventDone and OutputDone messages from subordinates, it sends message 24 up the chain to indicate it has completed simulation output. Note that the rightmost coordinator has bagged an external event, but it may not have been an imminent model. The chain of ProcessEventMessages starting with message 25 propagates down the hierarchy, giving each ModelCoordinator an opportunity to become imminent if it has received an external event that must be handled. By this mechanism, the rightmost model coordinator joins the model simulator as an imminent model.

4.3 Execute Transition

Messages 31-38 in Figure 1 represent simulation state transition. The previous flow of messages has generated two imminent models, the model simulator, which will execute internal state transition, and the rightmost model coordinator, which must handle and distribute the event message it has bagged. When the model coordinator receives message 32, it will handle its bagged external event by passing to an internal subordinate model (not shown in the diagram), which will itself then become immediately imminent. It indicates that it is immediately imminent by passing the time of its next transition as the current time in its TransitionDone message.

When the model simulator receives message 35, it calls the internal state transition function of the DEVS model, which will modify internal state, schedule future events, and schedule model output as necessary before sending message 37 to

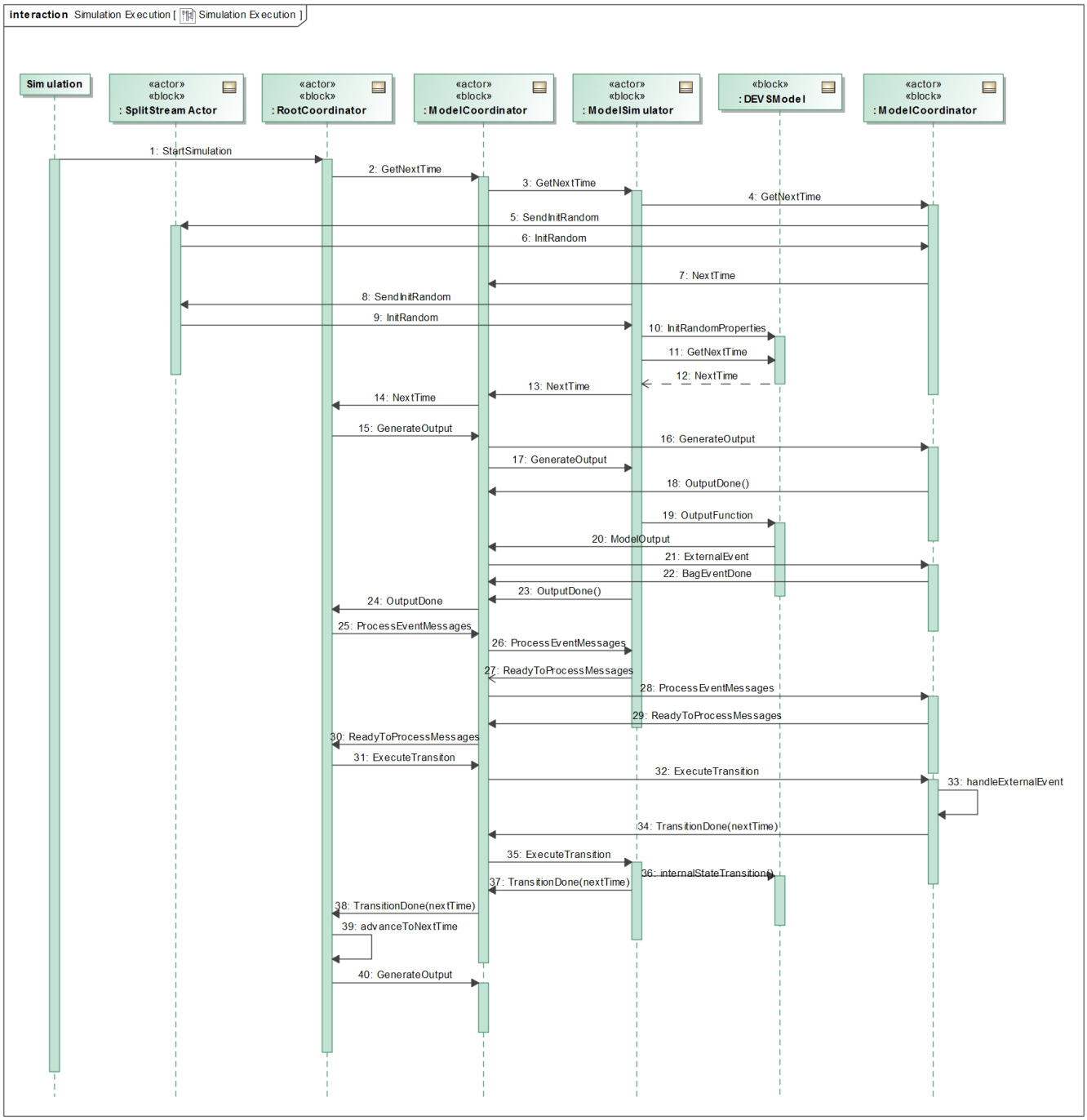

Figure 1. Sequence of communications between actors in a DEVS-DMF simulation run.

its parent indicating completion of state transition. The TransitionDone message also communicates the time of a model's next state transition to its parent coordinator.

This TransitionDone message plays another important role in the DEVS-DMF architecture. A state transition function may, in its calculations, create new actors, asynchronously call external microservices, and aggregate responses. Its parent coordinator will asynchronously wait for all of this to complete because it has not yet received the TransitionDone message from its imminent subordinate. These messages enable distributed and asynchronous execution of the simulation model.

When notified that all transitions are complete. The root coordinator will advance time to the next scheduled transition time of subordinate models. Note that, in this case, output messages and resultant event messages have caused an internal model in the rightmost model coordinator to become immediately imminent, so the global clock will not advance. The root coordinator will start the process over at the current time by sending a GenerateOutput message down the chain. The generate output and execute transition phases of the simulation will repeat until the termination criteria of the simulation have been met.

5. DEVS-DMF TESTING AND IMPLEMENTATION

Our research group has two implementations of DEVS-DMF to date. One is an implementation of a parallel computing benchmark in order to see if appropriate speedup takes place when scaling up by adding processors. The second implementation involves the integration of existing disparate models to support analysis of weapons systems.

5.1 Parallel Benchmarking

In order to test the parallel creation and execution of actors, the research team created a DEVS-DMF adaptation of the PHOLD benchmark for parallel discrete event simulation[5]. In the model, 100 actors were placed on a two dimensional grid. A "bean" is placed in each cell. At each time step, the bean randomly jumps to one of its adjacent cells. The computation expense for determining the jump destination is artificially set to 80 milliseconds, representing a moderately expensive state transition computation. A run on a virtual machine with one core took 80.152 seconds. The simulation was executed with 4 cores, taking 22.557 seconds.

In interpreting the results, it is important to know how Akka schedules actor execution. The default executor has a thread pool for the actor system. Actors consume no resources when they have no messages to process. When an actor receives a message, it is allocated a thread from the thread pool to process the message. Upon completion, it returns the thread to the pool. In this fashion, imminent actors will request threads necessary for transition at the current time. Using this execution scheme, the 4 available cores were able to achieve a speedup factor of nearly 4. The benchmarking shows, as expected, that significant speedup is achieved by adding processors, but dependencies in the simulation do not allow a speedup equal to the number of processors.

5.2 Weapons Analysis Simulation

Analysis of weapons broadly encompasses four qualities - weapon attributes, ammunition attributes, shooter attributes, and the weapon systems effects on the target. The effectiveness of one weapon system compared to another is normally evaluated based on outputs of physics, engineering, or probabilistic model. The Army Materiel Systems Analysis Activity uses these models individually to compare, for example, one weapon's accuracy to another or the damage effects of one munition to another. They also use combat simulations to compare operational effectiveness of a squad with one weapon system to its effectiveness with another. However, the space between individual physics-based models and operational models is a gap in the organization's modeling capabilities. It cannot model the interactions between characteristics such as weapon accuracy and rate of fire without running a full combat simulation. However, combat simulations introduce many confounding factors, such as terrain and tactics, which may mask the impact of the weapons characteristics of interest. This approach is often too complex and time consuming, and it does not support large experimental designs.

Analysts can run the models individually, but the output of one model must be manually transformed so that it can be sent to the next model. Again, this is time consuming and does not support large experimental designs. To address this problem, the modeling team developed the individual physics based models as microservices available in a cloud infrastructure. They then integrated these services using the DEVS-DMF framework. This provides the analyst more flexibility to analyze a system at the appropriate level.

The SysML internal block definition diagram in Figure 2 shows the weapons effectiveness simulation architecture. The simulation models one weapon firing at one target until the target is incapacitated. The architecture consists of a top-level coordinator that has one coupled model and one atomic model inside. The atomic model is a target. The soldier coordinator is a coupled model that has both a sensor atomic model and a rifle atomic model. The sensor model executes a target acquisition microservice until the target is acquired, generating an output to the soldier coordinator that routes it to the rifle model as an event. The rifle model uses a rate of fire microservice and an accuracy microservice to generate munition fired outputs, decrementing its ammunition count internal state variable each time. The soldier coordinator routes the munition fired output to the top-level coordinator, which routes it to the target model. Upon receipt, the target model uses round location data from the munition fired event, a target geometry microservice, and a target damage microservice to determine whether the round hits the target and, if so, the damage. When the target is incapacitated, it generates a target incapacitated output that is routed to the rifle to stop firing, terminating the simulation.

Another key aspect of DEVS-DMF is the degree to which simulation management and boilerplate code development can be handled by the framework. The framework itself handles scheduling and execution via the parallel DEVS implementation. Once a developer designs the simulation architec-

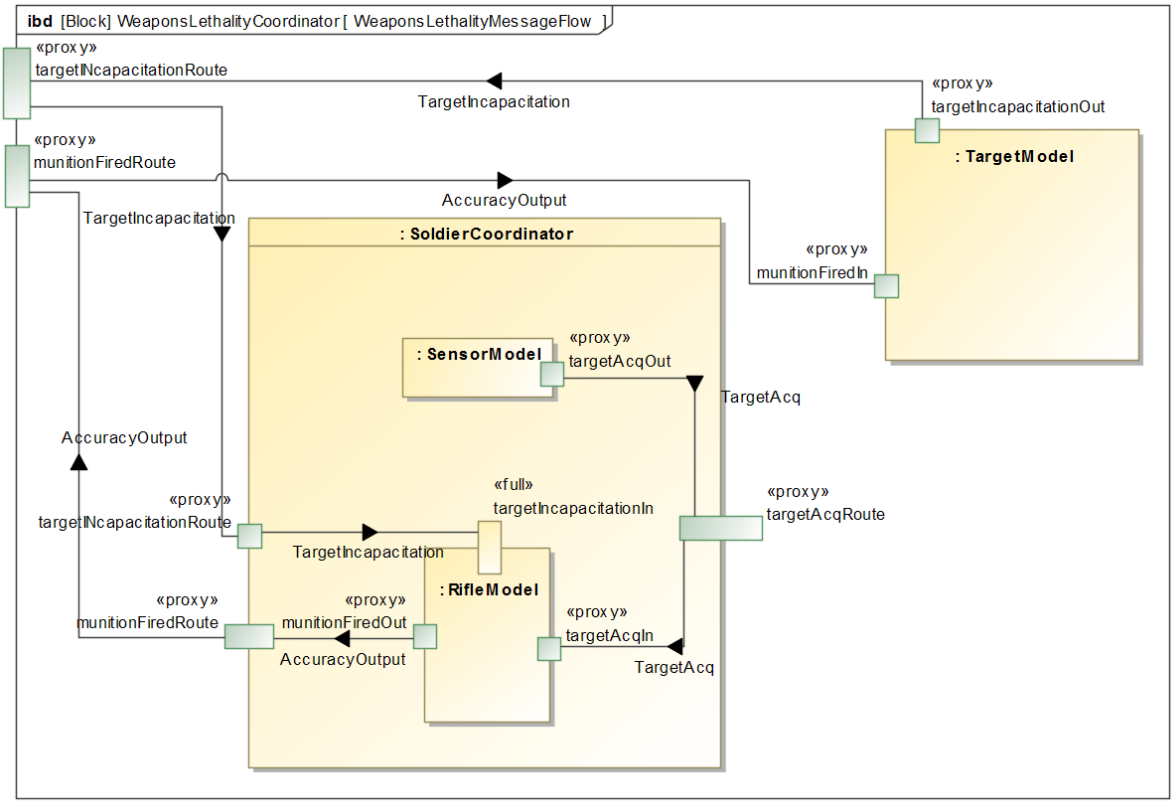

Figure 2. Flow of output messages and events in weapons analysis simulation.

ture, such as the one shown in Figure 2, code generation can write source files for actors and classes to manage the simulation. The developer is left to focus on writing code for state transitions and for the routing of messages between models by internal and external event handlers in separate implementation files.

The analyst can dynamically scale the analysis to include any number of these models as the problem set dictates. An analyst could potentially analyze systems starting at the single model level, such as accuracy, and build up to an operational model including terrain, tactics, and cognition by simply adding models and services together. The capability would allow the analyst to conduct a robust analysis of each component, the system, its effect on a small unit, and its effect on a larger force to inform decision makers about what matters at each level.

6. CONCLUSIONS AND FUTURE WORK

DEVS-DMF provides significant capabilities to those interested in encapsulating simulation components as services and deploying them to containers. These include:

DEVS and the Actor Model The underlying conceptual model of DEVS-DMF takes advantage of the synergy between DEVS and the actor model of computation. An actor's concurrency model relieves the programmer from having to explicitly manage shared state.

Partitioning The challenge of partitioning the parallel simulation is handled by encapsulating state transitions within discrete DEVS atomic models.

Separation of Concerns The DEVS-DMF application developer can focus on the logic of state transitions within models and passing of messages between models, leaving the framework to handle simulation structure and synchronization.

Flexibility DEVS models can be flexibly deployed within the same java virtual machine or anywhere on the network, interacting only with messages. This asynchronous message-based integration eases further integration with existing microservices deployed within the organizations infrastructure.

Development of the DEVS-DMF framework into phase II continues. The following additional requirements are under development.

- Web-based execution and visualization of state data and property data.
- A graphical model and user interface to compose models, events, and data into a working simulation.
- Parallel and distributed implementation for using cloud-based containers.

- Support for implementing microservices or functions via network messaging.
- Design and analysis of experiments.
- Persistent storage for simulation events and state trajectory
- A capability for heuristic simulation optimization.

7. ACKNOWLEDGMENTS

The DEVS-DMF program would like to thank the Army Research Lab Simulation and Training Technology Center, the US Army Corps of Engineers Institute for Systems Engineering Research, and the Army Materiel Systems Analysis Activity for their support. An additional thanks to the talented group of analysts and developers who make this program possible.

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the Department of the Army, DOD, or the U.S. Government.

REFERENCES

1. *Parallel Java Library*. https://www.cs.rit.edu/ ark/pj.shtml last checked Dec 6, 2015, January 2015.
2. Agha, G. A. Actors: a model of concurrent computation in distributed systems. Tech. rep., MIT Artificial Intelligence Lab, 1985.
3. Chow, A. C. H., and Zeigler, B. P. Parallel devs: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, Society for Computer Simulation International (1994), 716–722.
4. DâAngelo, G., and Marzolla, M. New trends in parallel and distributed simulation: From many-cores to cloud computing. *Simulation Modelling Practice and Theory 49* (2014), 320 – 335.
5. Fujimoto, R. M. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990), 23–28.
6. Fujimoto, R. M., Malik, A. W., and Park, A. J. Parallel and distributed simulation in the cloud. *SCS M&S Magazine* (July 2010).
7. Guan, S. *A Multi-layered Scheme for Distributed Simulations on the Cloud Environment*. PhD thesis, University of Ottowa, 2015.
8. Hewitt, C., Bishop, P., and Steiger, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, Morgan Kaufmann Publishers Inc. (San Francisco, CA, USA, 1973), 235–245.
9. IEEE-SA Standards Board. Standard for Distributed Interactive Simulation – Application Protocols. Technical Report IEEE 1278.1A1998, 1998.
10. IEEE-SA Standards Board. IEEE Standard for Modeling and Simulation M&S High Level Architecture (HLA) – Framework and Rules. Technical Report IEEE 15162000, September 2000.
11. Kaminsky, A. *Building Parallel Programs: SMPs, Clusters, and Java*. Cengage Course Technology, 2010.
12. Kewley, R. *DEVS Distributed Modeling Framework*. `https://github.com/rkewley/devsdmf` .
13. Kewley, R., MacCalman, A., McDonnell, J., and Hein, C. DEVS distributed parallel architecture for enterprise simulation. In *Proceedings of the 2015 Conference on the Theory of Modeling and Simulation*, Society for Modeling and Simulation International (April 2015).
14. Kewley, R., and Sapol, S. Executable Architecture for Systems Engineering – Distributed Modeling Framework. In *Proceedings of the 2014 Fall Simulation Interoperability Workshop*, no. 14F-SIW-058 (2014).
15. MacCalman, A., Kwak, H., McDonald, M., and Upton, S. Capturing experimental design insights in support of the model-based system engineering approach. In *Proceedings of the 2015 Conference on Systems Engineering Research* (2015).
16. Namiot, D., and Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.
17. Nutaro, J. On constructing optimistic simulation algorithms for the discrete event system specification. *ACM Transactions on Modeling and Computer Simulation 19*, 1 (January 2009), 1–21.
18. Object Management Group. *Systems Modeling Language*. `http://www.omgsysml.org/` last checked 13 Dec 2015, 2014.
19. Roberto Beraldi, Libero Nigro, A. O. Temporal uncertainty time warp: An implementation based on Java and ActorFoundry. *Simulation 79*, 10 (2003), 581–597.
20. Typesafe, Inc.,. *Akka Scala Documentation - Release 2.3.3*. `http://akka.io/docs/` last checked 13 December 2015.
21. Wampler, D. *Apache Spark and the Typesafe Reactive Platform: A Match Made in Heaven*. `http://typesafe.com/blog/` last checked 6 Dec 2014, July 2014.
22. Yoginath, S. B., and Perumalla, K. S. Efficient parallel discrete event simulation on cloud/virtual machine platforms. *ACM Transactions on Modeling and Computer Simulation 26*, 1 (2015), 5:1–5:26.
23. Ziegler, B., and Nutaro, J. What's the best possible speedup achievable in distributed simulation: Amdahl's law reconstructed. In *Proceedings of the 2015 Conference on the Theory of Modeling and Simulation*, A. e. a. D'Ambrogio, Ed., vol. 47 of *Simulation Series*, Society for Modeling and Simulation International (Alexandria, Virginia, 2015), 189–196.