Model Management and Execution in DEVS Unified Process

José L. Risco-Martín[1] and Saurabh Mittal[2]

[1]Universidad Complutense de Madrid, Spain
[2]The MITRE Corporation, USA

February 17, 2019

Abstract

Any Modeling and Simulation (M&S) endeavor comprise of two activities: modeling and simulation. Though each is an independent discipline in itself, often M&S is taken together as an integrated approach executed through software engineering practices. If not developed through a component-based methodology, it may result in various problems at later stages of any large scale M&S effort, for example, maintenance, integration, augmentation, verification and validation. In this chapter, we will describe the model engineering process for projects that subscribe to discrete event world-view. We will present model management from various perspectives as described in DEVS Unified Process (DUNIP) in Mittal and Risco-Martín's earlier work. After specification of model, we will then focus on the transparent execution of models in a net-centric System of Systems environment and present state-of-the-art in model and simulation interoperability within DUNIP.

1 Introduction

Cloud infrastructures provide rapid resource provision for on-demand computational requirements. Cloud simulation environments are largely client-server architectures with multiple slave nodes solving problems through Monte Carlo methods. A cloud simulation is not the same as a distributed simulation. A cloud implementation of an M&S application does not ensure that the infrastructure will scale and complexity inherent in distribution simulation is addressed. In addition, implementing distributed M&S services in cloud infrastructure is a non-trivial problem [2]. Having a cloud-based deployment does not guarantee that a distributed simulation infrastructure is a "given". Both have different architectures. However, cloud computing brings on-demand resources and technologies like virtualization and containerization. Incorporating cloud computing for distributed M&S infrastructure then is a logical thing to get the best of both

technologies and capitalize on the Moore's law with minimal increase in physical hardware costs.

In this article, we will describe a methodology to deploy a formal discrete event dynamic system simulation infrastructure based on Discrete Event Systems (DEVS) formalism, known as DEVS/SOA [13, 16] in a distributed cloud environment. DEVS is component-based M&S framework founded on mathematical Systems theory. DEVS also supports model continuity through a simulation-based development and testing life cycle [7]. This means that the mapping of high-level requirement specifications into lower-level DEVS formalizations enables such specifications to be thoroughly tested in virtual simulation environments in cloud environments before being easily and consistently transitioned to operate in a real environment for further testing and fielding.

A scalable M&S architecture has distinct modeling and simulation layers. In order to deploy in cloud environment, sufficient automation is needed at both the simulation layer and the modeling layer. This can now be achieved by current practices in DevOps implemented using Docker technology. DevOps, a recent buzzword, provides methodologies to automate developer operations, such as compiling, building, releasing, testing through executable scripts. DevOps has recently been applied to DEVS component-based models (as "nodes") [20]. This automated deployment of various "DEVS nodes" under a single administrative control is defined as a DEVS Farm. A DEVS Farm can be readily deployed and put to use for distributed simulation. A cloud insfrastructure is not the only mean to deploy a DEVS Farm. The High Level Architecture (HLA) can serve to perform these distributed simulations, as well. An example is given in another chapter presented in this book, entitled *DEVSim++ME: HLA-compliant DEVS Modeling/Simulation Environment with DEVSim++*, which presentes a model engineering environment with support for the development of HLA-compliant DEVS models for discrete event systems. Our chapter is more focused on a cloud-based M&S methodology and in the concept of *Simulation as a Service (SaS)*.

DEVS Unified Process (DUNIP) leverages the above advancements along with the foundational DEVS. It was conceptualized by Saurabh Mittal [13] during his doctoral work in collaboration with Jose L. Risco Martín and Bernard Zeigler [16]. From the original SOA-based implementation, it evolved in the last 10 years to include constructs like domain specific languages (DSLs), metamodeling, model-driven engineering, automated code-generation, model-based testing, test-suite generation and the more recent, microservices and containerization support. DUNIP has been applied to M&S of netcentric System of Systems engineering (SoSE) for its extensibility, modularity, flexibility and explicit semantics. This chapter provides an overview of the state-of-the-art on DUNIP. It will describe model management, model engineering and model execution mechanisms in DUNIP.

The chapter is organized as follows. Section 2 presents the discrete event world-view on why it is important to understand discrete event system in a formal way, especially when applying M&S to SoSE. Section 3 provides an overview of DUNIP. Section 4 describes the modeling infrastructure within

DUNIP through its two major elements: the DEVS Modeling language (DEVSML) and the improved DEVSML Stack Version 3.1. Section 5 provides the simulation infrastructure details in a Cloud environment with Docker containerization support. Section 6 discusses model engineering in DUNIP using meta-modeling concepts, followed by model integration and interoperability considerations in Section 7. Finally, the paper is concluded in Section 8.

2 Discrete-event world-view

2.1 Overview

A simulation is an imitation of some real thing, state of affairs, or process in action. The act of simulating something generally entails representing certain key characteristics or dynamic behaviors of a selected physical or abstract system.

Simulation is used in many contexts, including the modeling of natural systems or human systems in order to gain insight into their functioning. Other contexts include simulation of technology for performance optimization, safety engineering, testing, training and education. Simulation can be also be used as a prediction tool to show the eventual real effects of alternative conditions and courses of action.

Key issues in developing a simulation technology include acquisition of valid source information about the referent, selection of key characteristics and behaviors, the use of simplifying approximations and assumptions within the simulation model, and fidelity and validity of the simulation outcomes.

A computer simulation attempts to simulate an abstract model (that is computationally represented) of a particular system. A system is part of the real world under study and that can be identified from the rest of its environment for a specific purpose. Such a system is called a real system because it is physically part of the real world. The state of a system is defined as that collection of variables necessary to describe a system at a particular time, relative to the objectives of a study.

Systems may be categorized in two types, discrete and continuous. A discrete system is one for which the state variables may change only at discrete values of time. It is also known as discrete-time system. A continuous system is one whose state is capable of changing at any instant of time. It is also known as continuous-time signal system. Few systems in practice are wholly discrete or wholly continuous, but it will be usually possible to classify a system as being either discrete or continuous [10]. In this chapter we are focused on discrete systems.

Continuing with the notion of system, a few things are common at the systems level: (1) there is a large number of components, (2) hierarchy is used to manage the large number of components and complexity, (3) interactions among components play a vital role in system's behavior, (3) a system has a boundary, (4) a system manifests outward behavior and internal states and (5) a system interacts with the environment.

A netcentric system is a system that utilizes standards to integrate and operate in a network centric environment. The network, which can be managed through a cluster, a data center, or the Cloud, is the underlying communication mechanism. Usage of widely adopted standards facilitates integration and interoperability.

At the complex systems level, things become a bit more complicated in the sense that the complex systems is inherently dynamic due to a lot of moving parts. A system may transform into a complex system. A complex system exhibits: nonlinear behavior, dramatic changes, low predictability, etc.

To model and simulate such a variety of systems (complex, centralized or distributed), recently identified as an open system concept in [18], the DEVS specification has been selected as the heart of DUNIP. DEVS is a theoretical framework to define, implement and simulated such heterogeneity in a consistent way. In the following, we briefly describe this formalism.

2.2 The Discrete Event System Specification

DEVS is a general formalism for discrete event system modeling based on Set theory [33]. The DEVS formalism provides the framework for information modeling which gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. Once a system is described in terms of the DEVS theory, it can be easily implemented using an existing computational library. The parallel DEVS (PDEVS) approach was introduced, after 15 years of the inception of Classic DEVS [32]. Currently, PDEVS is the prevalent DEVS, implemented in many libraries. PDEVS accounts for the confluent condition i.e. a time instant at which both internal and external event becomes imminent. In the following, unless it is explicitly noted, the use of DEVS implies PDEVS.

DEVS enables the representation of a system by three sets and five functions: input set (X), output set (Y), state set (S), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), output function (λ), and time advance function (ta).

DEVS models are of two types: atomic and coupled. Atomic DEVS processes input events based on their model's current state and condition, generates output events and transition to the next state. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings.

Particularly, an atomic model is defined by the following equation:

$$A = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \delta_{\text{con}}, \lambda, ta \rangle \quad (1)$$

where:

- X is the set of inputs described in terms of pairs port-value:
  $\{p \in IPorts, v \in X_p\}$.

- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$.
- S is the set of sequential states.
- $\delta_{\text{ext}} : Q \times X^b \to S$ is the external transition function. It is automatically executed when an external event arrives to one of the input ports, changing the current state if needed.
  - $Q = (s, e)s \in S, 0 \leq e \leq ta(s)$ is the total state set, where e is the time elapsed since the last transition.
  - X^b is the set of bags over elements in X.
- $\delta_{\text{int}} : S \to S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S.
- $\delta_{\text{con}} : Q \times X^b \to S$ is the confluent function, subject to $\delta_{\text{con}}(s, ta(s), \emptyset) = \delta_{\text{int}}(s)$. This transition decides the next state in cases of collision between external and internal events, i.e., an external event is received and elapsed time equals time-advance. Typically, $\delta_{\text{con}}(s, ta(s), x) = \delta_{\text{ext}}(\delta_{\text{int}}(s), 0, x)$.
- $\lambda : S \to Y^b$ is the output function. Y^b is the set of bags over elements in Y. When the time elapsed since the last output function is equal to $ta(s)$, then λ is automatically executed.
- $ta(s) : S \to \Re_0^+ \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle \tag{2}$$

where:

- X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$.
- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$.
- C is a set of DEVS component models (atomic or coupled). Note that C makes this definition recursive.
- EIC is the external input coupling relation, from external inputs of M to component inputs of C.
- EOC is the external output coupling relation, from component outputs of C to external outputs of M.
- IC is the internal coupling relation, from component outputs of $c_i \in C$ to component outputs of $c_j \in C$, provided that $i \neq j$.

Given the recursive definition of M, justified by closure under coupling [32], a coupled model can itself be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

3 DEVS Unified Process (DUNIP)

3.1 Overview

DUNIP is based on an open systems concept. An open system is a dynamical system that can exchange energy, material and information with the outside world through its reconfigurable interfaces over a period of time. An open system also possesses the capability to form complex hierarchical structures enabling them to compete and cooperate at the same time. In fact, the mechanism to reorganize in a hierarchical structure is one of the basic requirements to manage complexity. The open systems are also characterized by emerging behavior and evolving structure.

In order to have an executable adaptive System of System (SoS) model, DUNIP must provide capabilities to model an open system. In addition, a process also needs to be defined that allows the development of an executable open system. Much of the open system development hinges on the variable structure capability within a component-based system. Desired characteristics of an open systems modeling framework are the ability to add or remove hierarchical components, change connections among components and lastly, modify the behavior of a component as it evolves per its surroundings. While the first two capabilities are structural in nature and have been documented in DEVS literature, the third one is behavioral modification at runtime. This capability is the most difficult to achieve. The DEVS open systems approach underlying DUNIP gives it strong formal foundation to develop M&S complex systems software capable of designing emergent behaviors [14].

In an SoS, systems and/or subsystems often interact with each other because of interoperability and integration requirements. These interactions are achieved by efficient communication among the systems using either peer-to-peer communication or through central coordinator in a given SoS. Since the systems within SoS are operationally independent, interactions among systems are generally asynchronous in nature. A simple yet robust solution to handle such asynchronous interactions (specifically, receiving messages) is to throw an event at the receiving end to capture the messages from single or multiple systems. Such system interactions can be represented effectively as discrete-event models. In discrete-event modeling, events are generated at random time intervals as opposed to some pre-determined time interval seen commonly in discrete-time systems. More specifically, the state change of a discrete-event system happens only upon arrival (or generation) of an event, not necessarily at equally spaced time intervals. To this end, a discrete-event model is a feasible approach in simulating the SoS framework and its interaction. There are many discrete-event simulation engines that can be used in simulating interaction in a heterogeneous mixture of independent systems. The main advantage of DEVS is its effective mathematical representation and its support to distributed simulation.

DEVS formalism has been in existence for over 40 years. It has been applied to multiple domains and many of the continuous, discrete or hybrid formalisms

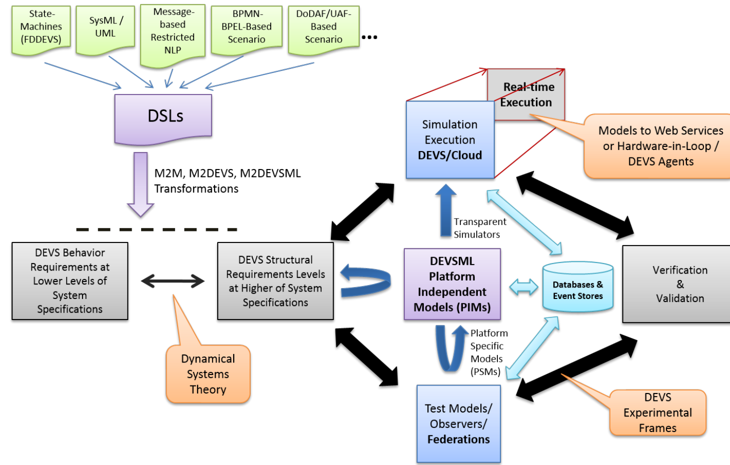

Figure 1: The DEVS Unified Process (improved from [18])

can be reduced to the DEVS formalism [32]. DEVS is based on Systems theory with its hierarchy of system specifications and closure under coupling properties. DUNIP is focused on DEVS and is the consummation of how DEVS can be applied to System of Systems design and analysis in full systems engineering life cycle setup [13]. DUNIP is not a single concept but an integration of various concepts that have been developed over the years in DEVS research. These concepts have now evolved into an integrated process that facilitates complex systems modeling and simulation. Combining the Systems theory, M&S framework and model-continuity principles, it leads naturally to a life-cycle development process, originally referred as Bifurcated Model-Continuity Based Life Cycle Methodology [31]: a precursor to DUNIP.

DUNIP is a universal process and is applicable in multiple domains. However, the understated objective of DUNIP is to incorporate DEVS formalism as the binding factor at all phases of this development process. Figure 1 illustrates the DUNIP, adapted from [18].

The important concepts and the process within DUNIP are listed below:

1. *Requirements specification using Domain Specific Languages (DSLs)*: Domain specific languages (DSLs) are used to specify system requirements and definitions. This item is described in Section 4.

2. *Platform Independent modeling at lower levels of systems specification using DEVS DSL*: This is performed through the DEVS Modeling Language (DEVSML), which is also presented in Section 4.

3. *Model Structures at higher level of System resolution using System Entity Structures (SES)*: This item is focused on the role of System Entity Structures [34] at higher levels of systems specification and a model-based

repository framework in which components stored in a repository can be used for systems development, which is briefly mentioned in Section 4.

4. *Platform Specific Modeling (PSM), i.e., DEVS implementations on different platforms*: Sections 4 and 5 show how platform independent DEVS models can be implemented in a platform specific language such as JAVA, C# or C++.

5. *Netcentric execution in a distributed setup*: Section 5 presents several frameworks for DEVS execution, with details on the DEVS simulation architecture, distributed message management and cross-platform execution of DEVS PIMs. It will show how to define a DEVS/Cloud simulation engine based on DEVS/SOA (and supported by xDEVS [23]) to achieve distributed execution.

6. *Automated Test Model generation using DEVS PIMs*: Automated generation of DEVS observers and test agents from DEVS platform independent models is discussed at length in [18]. It describes how DEVS DSL plays a critical role in achieving this capability.

7. *Interfacing of models with real-time systems*: DEVS can act as a production system and can interface with live services, hardware-in-the-loop and Live, Virtual and Constructive environments [12].

8. *Verification and Validation(V&V)*: The subject of V&V is a critical aspect in developing any theory or the modeling thereof. Without valid models, the theory cannot be tested. Without verified models, model's correctness cannot be ensured. Experimental Frame design for V&V is addressed in [18].

4 Modeling infrastructure

DUNIP defines the modeling infrastructure (as well as the simulation infrastructure) through the DEVSML framework (currently in its version 3.0 [20]). The DEVSML 3.0 framework has two pieces: the language and the stack. In the following we describe these two pieces.

4.1 DEVSML: A language

A DSL is a dedicated language for a specific problem domain and is not intended to solve problems outside it. For example, HTML, Verilog, VHDL, etc., are DSLs for very specific domain. A DSL can be textual or a graphical language or a hybrid one. A DSL builds abstractions so that the respective domain experts can specify their problem well suited to their domain understanding without paying much attention to the general purpose computational programming languages such as C, C++, Java, etc., which have their own learning curve. The notion of domain specific modeling arises from this concept and the DSL designers are

package	import	entity
extends	coupled	models
interfaceIO	couplings	atomic
ic	eoc	eic
vars	state-time-advance	state-machine
start in	confluent	deltint
deltext	outfn	sigma
continue	reschedule	ignore-input
input-only	input-first	input-later
infinity	int	double
String	boolean	input
output	S:	S'':
this	X:[]	Y:[]

Table 1: DEVSML Keywords [18]

tasked with creating a domain specific modeling language. If a DSL is also meant for simulation purposes, then one more task of mapping a specific DSL to a general purpose computational language is also on the cards. There are many DEVS DSLs that implement a subset of rigorous DEVS formalism. One example of DEVS DSL is DEVSpecL [6], built on Backus-Naur Form (BNF) grammar. DSL writing tools like Xtext, Ruby, etc., focusing directly on the Extended BNF (EBNF) grammar provide a much easier foundation to develop the Abstract Syntax Tree (AST) for Model-to-Model (M2M) transformations. The rich integration and code generation capabilities with open source tools like Eclipse give them strong acceptance in the software modeling community.

The DEVSML standard has coexisted through different DEVS DSL, like the DEVS Modeling language [15] based on Finite Deterministic DEVS [8], an earlier developed XML-based XFD-DEVS [21] and an expanded specification of XFD-DEVS in [18]. Like any language, the DEVSML also has certain reserved keywords, shown in Table 1. A DEVSML file is of the extension *.fds* and the specification language contains three primary element types i.e. the Atomic, the Coupled and the Entity. While the atomic DEVS formalism has a notion of ports (input and output), the DEVSML language has a notion of messages specified as Entity structures that are eventually transformed to port definitions. The DEVSML grammar is specified using Eclipse Xtext EBNF notation and is available in [15, 18].

Once the DEVSML language has been designed, the implementation of a DEVSML editor is quite straightforward, using for example, Xtext DEVSML editors in eclipse. The current implemented framework, named DEVSML Studio [19], allows the user a complete validation mechanism to check both DEVSML structure and behavior, as well as assistance with some automatic tasks, like dynamic code generation. Figure 2 shows the DEVSML Studio atomic model rendition and Figure 3 shows the simulation run in the Eclipse DEVSML Studio [11].

Figure 2: DEVSML Studio showcasing the textual and autogenerated graphical DEVS state machine in Eclipse Integrated Development Environment [11]

Figure 3: DEVSML Studio showcasing the simulation log in a textual and autogenerated graphical UML Sequence Diagram in Eclipse IDE [11]

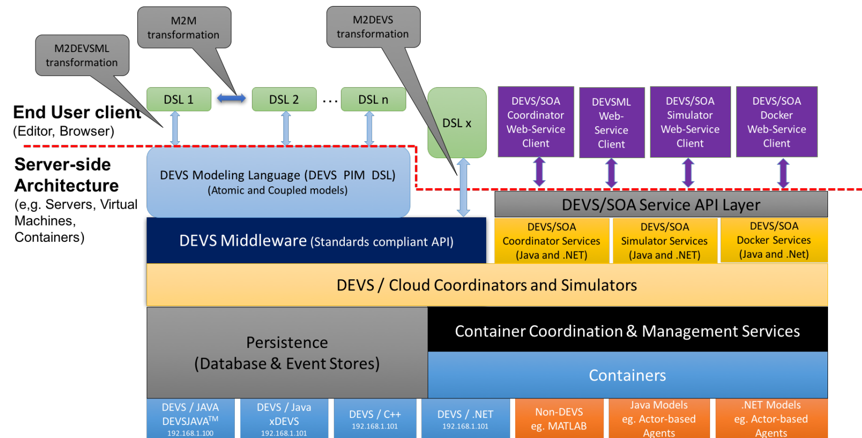

Figure 4: DEVSML Version 3.1 stack employing model-to-DEVS transformations for Model and simulator transparency in DEVS/Cloud (adapted from [20])

4.2 DEVSML stack

The purpose of the DEVSML stack is to integrate the transparent modeling framework with the inclusion of DSLs (like the DEVSML) through various transformations. DEVSML stack describes how platform independent DSLs can be transformed in this framework and eventually become operational in conjunction with the DEVS formalism.

One of the greatest advantages of DEVS is that model and simulator are completely decoupled. It allows the modeler to construct model in a platform of his choice. The ability to execute DEVS models in multiple platforms and languages has already been achieved and demonstrated, as summarized in [22]. In all these cases, the design of DEVS models was dependent on the language where the underlying assumption always has been "everything is an Object". In the new DEVSML framework, this concept has evolved to "everything is a model" [18] and there are two choices. Either the DSL designer takes the DSL directly to the execution code which involves no transformations but only code generation to the native programming language or he works with an existing framework that guarantees execution in formal systems theoretic way. If the DSL designer opts for the second option, choosing DEVS as a framework is recommended due to its rich history of model specification and simulator development. This ability provides a scalable solution, and has inherent advantages for programmatic integration and M&S interoperability. Having a process to transform any DSL to DEVS components, especially to the DEVSML platform-independent specification, then has obvious advantages.

DEVSML 3.0 Stack was proposed in [20], is improvised as Version 3.1 in Figure 4. Starting at the bottom of the Figure 4, the execution layer of the DEVS/Cloud simulator is built upon DEVS simulators in native languages (e.g.

C++, .NET, Java, etc.) that may get deployed as individual Docker containers as detailed in Section 5. The containers are built from the container images (templates) stored in the persistence layer. The next layer is the distributed communication and coordination layer that manages the containerization. This layer also incorporates the Docker scripts that build containers. Above that is the DEVS/Cloud layer that implements DEVS coordinators and simulators to perform distributed simulations along with the required local databases for microservice implementation. Next is the DEVS Middleware and the DEVS/SOA Service layer that makes available the DEVS modeling and DEVS simulation services for multifarious clients. It is this Service/Middleware layer that enables the transparent M&S framework. Finally, at the top, we have the DSLs and various Service clients that utilize the DEVS M&S services. To achieve model interoperability, DEVS models can be encoded in any given language that conforms to DEVSML Application Programming Interface (API). Otherwise, DEVS wrappers can wrap the component's behavior as a DEVS model [12]. The coupled models are then specified using a platform neutral format, e.g., XML/JSON.

We need to make a clear distinction here that the DEVS Modeling "language" is a DEVS modeling specification language that is anchored to DEVS simulation layer using the simulation relation in DEVS Middleware API. Consequently, a DEVSML specified model is a *bona fide* DEVS executable. The idea of including other DSLs at the top layer of the stack was a major addition in DEVSML Stack Version 2.0., which also added three transformations at the top layer:

1. Model-to-Model (M2M)
2. Model-to-DEVSML (M2DEVSML)
3. Model-to-DEVS (M2DEVS)

The key idea being: domain specialists (the end-user) need not delve in the DEVS world to reap the benefits of DEVS framework. The end-user as indicated in Figure 4 will develop models in their own DSL and the DEVS expert along with the DSL designer will help develop the M2M and M2DEVSML transformation to give a DEVS backend to the DSL models. While M2DEVSML transformation delivers an intermediate DEVS DSL (the DEVSML DSL), the M2DEVS transformation directly anchors any DSL to platform specific DEVS. On a reverse note, a DEVS expert is ideally suited to develop DSLs in other domains as developing transformations like M2DEVS and M2DEVSML need not be negotiated with the DSL expert. A DEVS expert with DEVSML skill set can perform a dual job of both the DSL and DEVSML expert.

The addition of M2M, M2DEVSML and M2DEVS transformations to the DEVSML stack adds true model and simulator transparency to a net-centric M&S distributed infrastructure. The transformations yield platform independent DEVS models (PIMs) that can be developed, compared and shared in a collaborative process within the domain. Working at the level of DEVS DSL

allows the models to be shared among the broad DEVS community that brings additional benefits of model integration and composability. DEVSML 3.0 stack allows DSLs to interact with DEVS middleware through an API. This capability enables the development of simulations that combine and execute DEVS and non-DEVS models [22]. This hybrid M&S capability facilitates interoperability. The scale is provided by the underlying distributed (or Cloud) infrastructure that is largely made of virtualization technologies and utilizes platform-as-a-service (PaaS) capabilities provided by containerization, as described in the next section. To support containers, a persistence layer is added in the proposed DEVSML Version 3.1 to account for databases and event stores. Database may store various container images and event stores help preserve runtime state in a microservices-based execution [20].

5 Execution (simulation) infrastructure

The DEVSML execution infrastructure is mainly based on the Cloud capabilities provided by one of its simulation engines, following the scheme of the DEVS Virtual Machine proposed in [18]. Then, using containers, a swarm of DEVS/Cloud simulators can be created, ready to perform distributed simulations.

Any microservices architecture is primarily an orchestration of stateless services. In a component-based M&S framework such as DEVS, a component-model has to be transformed to a stateless service and addresses two fundamental microservice architecture requirements: distributed data management and shared event stores. In the following, we discuss how the above aspects are handled at the *modeling* and *simulation* execution layers such that the model's state and inherent information can be externalized.

5.1 Modeling layer implementation

A DEVS model consists of ports (input and output), states and state-variables (including model name, current state, next state, time-of-last-event, time-of-next-event and elapsed-time) and the four characteristic functions:
$(\delta_{ext}, \delta_{int}, \delta_{con}, \lambda(s))$.

In a microservices-based rendition of a DEVS model, we have to partition these elements into the two buckets (of distributed data management and event stores) to achieve scalability through stateless execution of a modular DEVS component. The DEVS state-machine needs to be separated with the operations on the state-variables inside the DEVS atomic model component. The first bucket is the model's state, that is stored in state-variables, which have to be serialized in a local database for that model. This implies that all the operations on these variables through the four characteristic functions will be through the accessor functions i.e. *get* and *set*. The second bucket is of the shared event store that is used for the *input* X and *output* Y sets. The X and Y sets are transformed into declarative events (that may be implemented as a complex data-type) into a shared Event store. Likewise, the DEVS atomic

simulator components are partitioned as well. Table 2 shows the partiontioning of atomic DEVS elements (both modeling and simulation layers). Table 3 shows the coupled DEVS elements partitioning. The above partitioning allows the model to become stateless, which then can be containerized.

5.2 Simulation layer implementation

The simulation layer architecture will focus on the simulator and coordinator execution and how they implement the DEVS simulation protocol in an abstract-time manner. The architecture again has to account for the above two requirements: distributed data management and event stores. The DEVS simulation protocol defines the relationship between the model and its underlying simulator. The application of microservices provides resiliency at both the model and the simulator levels, to the effect that a large number of simulator instances with corresponding model instances can now be created when the data-exchange between the model and the simulator is accurately partitioned (as in Tables 2 and 3).

Table 2: Partitioning Atomic DEVS M&S Elements for Microservices implementation

Atomic DEVS model Specification	M&S Element	Data Management	Event Store	Comments
X	Model		x	Input events
Y	Model		x	Output events
$name$	Model	x		Model name
tl	Simulator	x		Time of last event
tn	Simulator	x		Time of next event
$phase$	Model	x		Current phase of the model
$v1, v2..vn$	Model	x		Values and their data types

We shall illustrate the microservices-based DEVS/Cloud simulation architecture implementation with the help of the classsic EF-P example (Figure 5). EF-P model contains two components: the Experimental Frame (EF) - Processor (P) models [18]. The hierarchical EFP model in Figure 5.a is flattened towards a Generator - Processor - Transducer (GPT) model depicted in Figure 5.b. Next, according to Figure 5.c, each of the three submodels in GPT, is mapped to a separate DEVS/Cloud node (container). Although, they all can also belong to a single container. One node can thus contain one or more DEVS models with their corresponding local databases to store model's data (Column 3 in Table 2). The node is selected through the simulation configuration file. For illustration purposes, Figure 5.c shows three nodes each corresponding to a single sub-model in GPT. One simulator is then created for each atomic model.

Table 3: Partitioning Coupled DEVS M&S Elements for Microservices implementation

Coupled DEVS model Specification	M&S Element	Data Management	Event Store	Comments
X	Model		x	Input events
Y	Model		x	Output events
$M1, M2..Mn$	Model	x		Sub-component names
I_Z	Model	x		Set of influencers
Z_{i-d}	Model/ Simulator	x		Mapping of influencer outports to influencee's inports

Due to the symmetrical architecture, each node also contains a blueprint of a coordinator. The simulation configuration file designates one of the nodes as the root coordinator. At the end, models' state and output events are always managed by their corresponding simulators, using the local databases and global event stores, respectively. Model output events are propagated through synchronous communication between the model-simulator pair, using the aforementioned `set` and `get` accessors. It should be highlighted that if the DEVS model is not flattened in the simulation configuration file, then the coupled information in Table 3 is stored as well.

The event store is implemented using various event cloud technologies such as Esper and provides a means to perform model-integrated systems engineering in which modeling and simulation itself is a part of the systems engineering [17]. The event stores are usually implemented as event clouds which lend themselves to Complex Event Processing (CEP) that can do streaming analytics as well as design monitors that can detect advanced spatio-temporal patterns between the messages flowing between the components. All the containers and global event store interface through a load balancer that divides the load on each container node through various load-balancing policies.

In the following, we briefly explain the automated deployment of a DEVS/Cloud node using Docker.

5.3 Implementation of the DEVS/Cloud support

A Docker Image is a read only template used to instantiate Docker containers. Each image is defined with several layers that compose the final image structure. The Docker registry also called Docker Hub is a Docker Image repository. Images can be downloaded or uploaded. The Docker Hub has a considerable amount of images ready to use. Finally, a Docker Container is the runtime component of the Docker Image. Multiple containers can be instantiated from the same Docker Image in an isolated context. Docker container can be run, started,

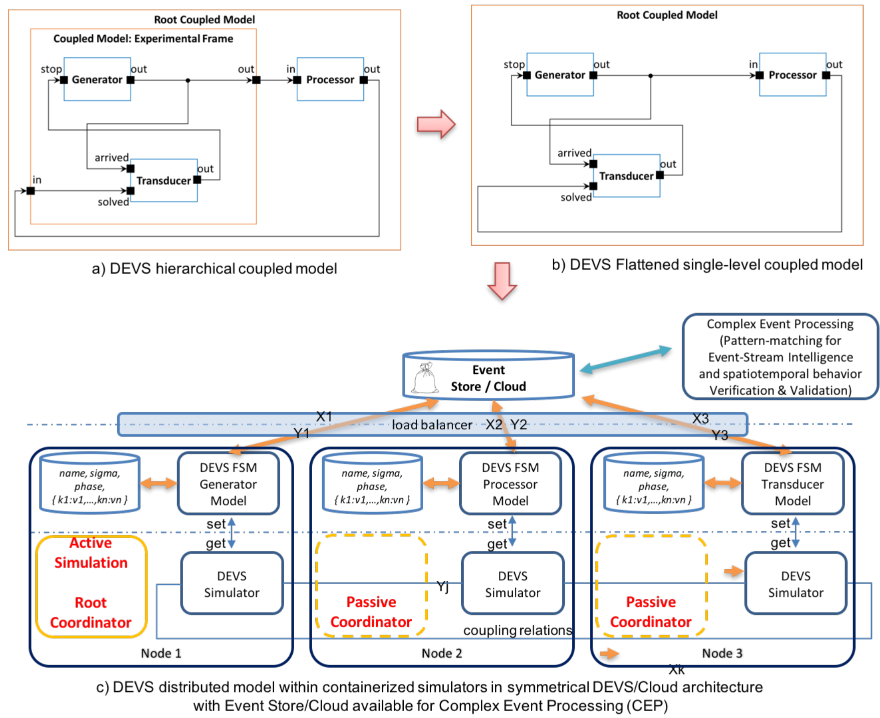

Figure 5: DEVS/Cloud container nodes within Microservices paradigm [20]

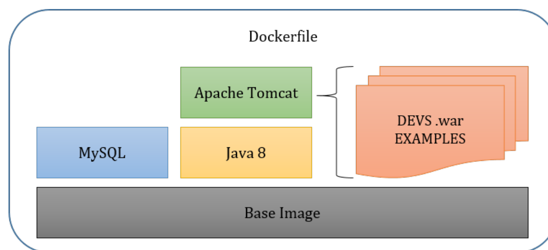

Figure 6: Full Dockerfile structure [20]

stopped, moved and deleted. To start, Docker must be installed in the host machine. For more information, the reader should be refer to the Docker official web page [3].

To build our DEVS/Cloud Dockerfile, we start from a DEVS/SOA complete WAR file [16], which includes the xDEVS simulation engine (with suppport for Simulators and Coordinadors as a Service) and several DEVS example models. The corresponding Docker image must then include a minimal runtime environment that contains the DEVS/SOA dependencies: Linux OS, Oracle Java 8 and an application service such as Apache Tomcat. Next, the .war file will be deployed into the webapps directory of Apache Tomcat.

Figure 6 depicts a Dockerfile structure. An excerpt of the source code of this file is shown in [20]. As Figure 6 shows, the Dockerfile must start with a base image. In our case, the base image is Ubuntu. Next, MySQL and Java 8 are added. The final step is to add Apache Tomcat. Finally, more security constraints can be added to the Dockerfile. Once the Dockerfile is completed, we can show and stop container through the `docker ps` and `docker rm` commands on the host OS that has the Docker daemon running.

Figures 5 and 6 show a transparent simulation infrastructure with DEVSML Stack Version 3.0 that incorporates container services for automated DEVS/Cloud deployments.

6 Model Engineering in DUNIP

6.1 Model-based/Model-driven Flavors

The 'model-based (MB)' and 'model-driven (MD)' terms and initials have been used in a variety of system and software related acronyms, such as MBD, MDSD, MDD, MDA, MBSE, MDE and many others. Although there is a consensus that these approaches suggest the systematic use of models as the primary means of a process and facilitate the use of DSLs, there is not a common understanding of the terminology [18]. The definitions of the frequently referred acronyms and the objectives of those approaches are summarized below:

1. MBE: Model-based Engineering (MBE) originated in the 1980's in parallel with the evolution of the Computer-Aided Design (CAD) and Model-Based Design (MBD) techniques. The main goal in MBE was to support the system development process during the design, integration, validation, verification, testing, documentation and maintenance stages [30, 29, 28].

2. MBSE: In systems engineering, the application of the MBE principles is called as Model Based Systems Engineering (MBSE) [29, 27, 35]. MBSE provides the required insight in the analysis and design phases, enhances better communications between the different participants and enables effective management of the system complexity.

3. MDE: Model-driven engineering (MDE) is a system development approach that uses models to support various stages of the development life cycle

[1, 25] and can be seen as a subset of MBE. MDE relies on technologies to automate model transformations thereby increasing productivity within MBE. It produces well-structured and maintainable systems because of its focus on formally defined models, metamodels, and meta-metamodels.

4. MDA: Model-Driven Architecture (MDA) is a software design and development approach that provides a set of guidelines for specifying and structuring models [4] relies on the Meta Object Facility (MOF) [5] integrate the modeling steps and provide the model transformations. MDA prescribes the use of metamodels and meta-metamodels for specifying the modeling languages without any necessity to be domain specific.

5. MDD or MDSD: The application of the MDE principles in software engineering is called MDD (Model Driven Development) or MDSD (Model Driven Software Development) [26]. The modern era of MDD started in the early 1990s and now offers a notable range of methods and tools. Different specifications such as MDA [4], MIC [9], Eclipse Modeling Project and Microsoft Software Factories are some of the conceptual applications of MDD principles.

6. MIC: Model Integrated Computing (MIC) refines the MDD approaches and provides an open integration framework to support formal analysis tools, verification techniques and model transformations in the development process [9]. MIC allows the synthesis of application programs from models by using customized Model Integrated Program Synthesis (MIPS) environments (e.g., Generic Modeling Environment [GME]). The meta-level of MIC provides metamodeling languages, metamodels, metamodeling environments and meta-generators for creating domain specific tool chains on the MIPS level.

MBE and MBSE utilize the systems verification and validation methodologies to the model dvelopment process relating back to the system requirements for systems test and evaluation. MDE is one area that serves both the software and systems engineering as it is domain independent. Model-Driven Systems Engineering (MDSE) gives MB/MBSE various model engineering, transformation and tools from MDE that speed up model development through the meta-modeling construct. The development of various editors based on MDE concepts involve advanced software engineering and code generation techniques. DUNIP (through its DEVSML Stack) is positioned as an MDSE that employs both the systems engineering and MDE paradigm in an agile manner [17].

6.2 Model Management in DUNIP

DEVSML and DUNIP are focused towards interoperability at the application level, specifically, at the modeling level and hiding the simulator engine as a whole. Our vision and solution development is along the lines of Model-as-a-Service (MaaS), Simulation-as-a-Service (SimaaS) and ultimately, DEVS-as-a-Service (DaaS). We would like the user or designer to code the behavior in any

of the programming languages, ideally a DSL of his choice and let the DEVSML 3.1 stack develop the transformations. The DEVSML Stack is responsible for taking a DSL or a coupled DEVSML model, integrating code within their DSLs and delivering us with an executable model that can be simulated on any DEVS platform (local, virtual, distributed or cloud).

The user can integrate his model from a model repository stored in any web location. It may contain publically available models of legacy systems or proprietary standardized models. Together they will provide more benefit to the industry as well as to the user, thereby truly realizing the model-based paradigm. In addition, the following aspects handle the model management in DUNIP:

1. Use of DSLs and guidance for model transformation: The DSLs can be platform-dependent or platform-independent. Through the model transformations (M2M, M2DEVSML and M2DEVS), any DSL can be brought in DUNIP. Using the underlying metamodel, a DSL can be mapped to the DEVSML metamodel.

2. Alignment with Systems Theory: The alignment reflects the presence of explicit interfaces between components and strong data-type exchange mechanisms between the interfaces. This facilitates unambiguous message exchange in an explicitly connected system.

3. Support for component reusability: DEVS and consequently, DUNIP, is a component-based framework. A component in DEVS is defined by two aspects: structure and behavior. Likewise, any DSL or a DEVS-wrapped component has an explicit interface and a defined behavior. This makes it available for its inclusion in a repository with a knowledge about the structure and behavior stored in the metadata for that component.

4. Code generation, execution and deployment mechanisms: DEVSML 3.1 Stack has a Service layer that keeps the code generation (for transformations), the execution layer (simulation), and the deployment (on cloud) transparent from the end-user and model workflows. The Service layer hence implements Maas, Simaas and DaaS.

7 Model integration and interoperability

Interoperability is a quality that denotes the ability of diverse independent systems to work together at a functional level. If two or more systems are capable of communicating and exchanging data between themselves to address a situation or solve a problem, the overall system manifests interoperablity between these systems. The word "system" can be a general concept for an organism, component, or an agent. Interoperability facilitates model extensibility and full integration. Thus, achieving a high degree of interoperability in simulation continues to be a prime objective within the research community and this age of heterogeneity. The main reason is to assist the confluence between the large

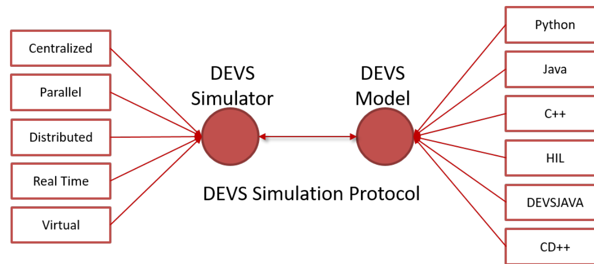

Figure 7: Conceptual Architecture of Standard

variety of legacy simulation frameworks and the latest simulation applications, not limited to augmented reality and virtual simulation.

During the last 10 years, a DEVS standard has been under development to support interoperability of DEVS models implemented in different platforms as well as with legacy simulations. Figure 7 illustrates an architectural approach proposed to accommodate the various combinations and permutations of possible application. The basic idea was to define two sets of interfaces; the DEVS model Interface and the DEVS Simulator Interface, as well as a DEVS simulation protocol that operates between the two.

DUNIP and its DEVSML construct (at both modeling and simulation layers) supports this conceptual architecture. At the modeling layer, DEVSML supports DEVSJAVA and xDEVS engines [23]. xDEVS contains wrappers for other DEVS M&S engines like DEVSJAVA, aDEVS or CD++. At the simulation layer, DEVSML implements xDEVS compatibility, with support for sequential, parallel and distributed simulations.

7.1 Integration at the modeling level

As mentioned above, DEVSML takes advantage of the existing DEVS model implementation interface in xDEVS [23]. To support the modeling requirements implemented as in Figure 7, the xDEVS API specifies an interface for both atomic and coupled models, which allows us to adapt the implementation to the DEVSML standard stated in Section 5. The compatibility of DEVS-to-DEVS models is tackled with the use of wrappers [16]. Figure 8 shows how the DEVS-to-Non-DEVS interoperability is solved. To start with, the DEVSML model interface is derived from the xDEVS interface (Figure 8). The xDEVS to DEVS mapping implements DEVS formalism specifications. These DEVS models can be implemented as full DEVS models or otherwise act as an adapter for non-DEVS integration, like for example Matlab integration [24].

Via the DEVSML simulator environment incorporated through the xDEVS framework (with all its sequential, parallel and distributed capabilities), we are capable of modeling and simulating atomic and coupled models that share the same semantics given by the DEVS mathematical specifications or through DE-

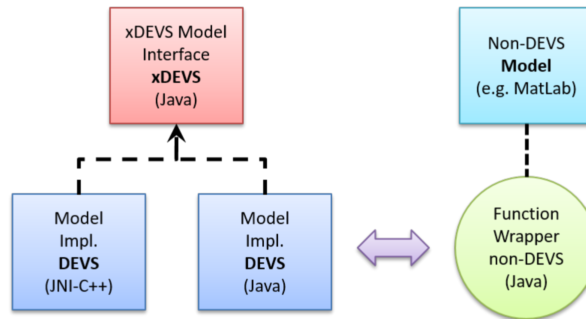

Figure 8: Integration at the modeling level

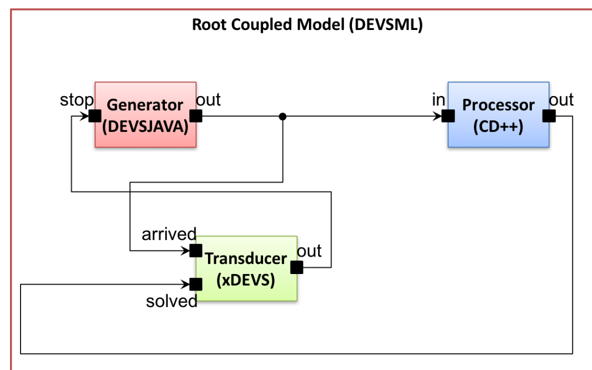

Figure 9: Cross Platform Coupled Model

VSML specifications. The models may differ in the computing environment implementation or deployment. As we have seen, implementations of both DEVS and non-DEVS "compliant" models (using an adequate wrapper for the case of non-DEVS models) that share a common DEVS interface can interoperate. As the DEVSML simulator remains unchanged, it is able to simulate and facilitate interoperable DEVS models implementations.

7.2 Integration at the simulation level

The above methodology can be extended for the xDEVS simulation framework (left side of Figure 7). As explained above, xDEVS provides DEVS-based simulators as services, which are based on standard communication technologies. Each atomic or coupled component may be implemented using different simulation engines, called platforms. Figure 9 depicts an example of multi-platform DEVS model. The DEVSML Stack act as the framework interoperating two different simulation platforms (Figure 5, Section 5).

8 Summary

The DEVS formalism, based on Systems theory, provides a framework and a set of modeling and simulation tools for SoSE [18]. A DEVS model is a system-theoretic concept specifying inputs, states, outputs, similar to a state machine. Critically different however, is that it includes a time-advance function that enables it to represent discrete event systems, as well as hybrids with continuous components, in a straightforward platform-neutral manner. DEVS provides a robust formalism for designing systems using event-driven, state-based models in which timing information is explicitly and precisely defined.

DUNIP is categorically designed to interface with service-oriented and cloud-based systems to bring an interoperable M&S environment that may include hardware-in-the-loop or software-in-the-loop (e.g. Service systems). Netcentric systems can be modeled effectively using the DEVS formalism and consequently can leverage DUNIP. To provide a brief overview of the current DEVS capabilities within DUNIP, Table 4 outlines how DEVS can provide solutions to the challenges in netcentric design and evaluation.

The realization of netcentric DUNIP has the following pieces:

1. DEVSML Stack: the central concept
2. Distributed simulation in Cloud environment over SOA
3. Netcentric DEVS Virtual Machine (both client and server)
4. Design, development and deployment of netcentric systems with DEVS
5. Containerization support for efficiency, scalability and scaleable deployment
6. Interfacing with Event Driven Architectures and Live, Virtual and Constructive environments that incorporate both hardware-in-the-loop and software-in-the-loop

DUNIP offers an integrated approach to bring in various DSLs in a methodical way to the DEVS ecosystem such that they could become a part of a larger SoS. This is very much a needed capability when it comes to complex multi-disciplinary M&S where models from multiple domains need to be brought in together. DUNIP incorporates the latest in MDE, DevOps and Cloud technologies to deliver a reliable M&S environment for rapid test and evaluation. Further improvements will be made as new technologies appear on the horizon.

DISCLAIMER The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author(s). Approved for Public Release. Distribution Unlimited. Case Number: PR_17-3254-10.

Desired M&S Capability for T&E	**Solutions Provided by DEVS Technology in DUNIP**
Support for executable architectures using M&S such as mission based testing for Cloud-based systems	DEVS Unified Process provides methodology and Cloud infrastructure for integrated development and testing [13]
Interoperability and cross-platform M&S using Cloud	Simulation architecture is layered to accomplish the technology migration or run different technological scenarios. Provide net-centric composition and integration of DEVS validated models using Cloud Computing.
Automated test generation and deployment in distributed simulation	Separate a model from the act of simulation itself, which can be executed on single or multiple distributed platforms. With its bifurcated test and development process, automated test generation is integral to this methodology
Test artifact continuity and traceability through phases of system development	Provide rapid means of deployment using model-continuity principles and concepts like "simulation becomes the reality" [7]
Real time observation and control of test environment	Provide dynamic variable-structure component modeling to enable control and reconfiguration of simulation on the fly. Provide dynamic simulation tuning, interoperability testing and benchmarking

Table 4: Mapping of M&S T&E capability requirements and DUNIP

References

[1] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20:36–51, 2003.

[2] Erdal Cayirci. Modeling and simulation as a cloud service: A survey. In *Proceedings of Winter Simulation Conference*, 2013.

[3] Docker. What is Docker?
https://www.docker.com/what-docker, 2017. Accessed Dec. 26, 2017.

[4] Object Management Group. Model Driven Architecture (MDA) Guide Version 1.0.1.
http://www.omg.org/mda/specs.htm, 2003.

[5] Object Management Group. Meta Object Facility (MOF) Core Specification, Version 2.0.
http://www.omg.org/spec/MOF/2.0, 2006.

[6] K. Hong and T. G. Kim. DEVSpecL-DEVS Specification Language for Modeling. *Information and Software Technology*, 48(4):221–234, 2006.

[7] X. Hu and B.P. Zeigler. Model Continuity in the Design of Dynamic Distributed Real-Time Systems. *IEEE Transactions on Systems, Man And Cybernetics - Part A*, 35(6):867–878, 2005.

[8] Moon Ho Hwang and Bernard P. Zeigler. Reachability graph of finite deterministic DEVS. *IEEE Transactions on Automation Science and Engineering*, 6(3):468–478, 2007.

[9] ISIS. Model Integrated Computing (MIC).
http://www.isis.vanderbilt.edu/research/MIC, 1997. Accessed Dec. 26, 2017.

[10] A. M. Law and W. D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, 2000.

[11] Dunip Technologies LLC. DEVSML Studio.
http://duniptechnologies.com/jm/downloads.html, 2015. Accessed Dec. 26, 2017.

[12] S. Mittal, M. Ruth, A. Pratt, D. Krishnamurthy, M. Lunacek, and W. Jones. A system of systems approach to integrated energy systems modeling. In *Summer Computer Simulation Conference*, Chicago, IL, 2015.

[13] Saurabh Mittal. *DEVS Unified Process for Integrated Development and Testing on Service Oriented Architectures*. PhD thesis, University of Arizona, 2007.

[14] Saurabh Mittal. Emergence in stigmergic and complex adaptive systems: A formal discrete event systems perspective. *Journal of Cognitive Systems Research*, 21:22–39, 2013.

[15] Saurabh Mittal and S. Douglas. DEVSML 2.0: The Language and the Stack. In *Proceedings of the 2012 Spring Simulation Multi-Conference*, 2012.

[16] Saurabh Mittal, José L. Risco Martín, and Bernard P. Zeigler. DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process. *SIMULATION*, 85(7):419–450, July 2009.

[17] Saurabh Mittal and José L. Risco-Martín. Model-driven systems engineering for netcentric system of systems engineering with devs unified process. In *Winter Simulation Conference*, 2013.

[18] Saurabh Mittal and José L. Risco-Martín. *Netcentric System of Systems Engineering with DEVS Unified Process*. CRC Press, 2013.

[19] Saurabh Mittal and José L. Risco-Martín. DEVSML Studio: A Framework for Integrating Domain-Specific Languages for Discrete and Continuous Hybrid Systems into DEVS-Based M&S Environment. In *Proceedings of the 2016 Summer Simulation Multiconference (SummerSim'16)*, 2016.

[20] Saurabh Mittal and José L. Risco-Martín. DEVSML 3.0 stack: rapid deployment of DEVS farm in distributed cloud environments using microservices and containers. In *Proceedings of the 2017 Spring Simulation Multi-Conference (SpringSim'17)*, 2017.

[21] Saurabh Mittal, Bernard P. Zeigler, and Moon Ho Hwang. XFDDEVS: XML-Based Finite Deterministic DEVS. http://www.duniptechnologies.com/research/xfddevs/, 2012.

[22] Alejandro Moreno, José L. Risco-Martín, Eva Besada, Saurabh Mittal, and Joaquín Aranda. DEVS/SOA: Towards DEVS Interoperability in Distributed M&S. In *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 144–153. IEEE, IEEE Computer Society, 2009.

[23] José L. Risco-Martín, Saurabh Mittal, Juan Carlos Fabero, Marina Zapater, and Román Hermida. Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION*, 93(6):459–476, 2017.

[24] José L. Risco-Martín, Alejandro Moreno, Joaquín Aranda, and J. M. Cruz. Interoperability between DEVS and non-DEVS models using DEVS/SOA. In *SpringSim'09: Proceedings of the 2009 spring simulation multiconference*, pages 1–9, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[25] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:25–31, 2006.

[26] M. Volter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.

[27] Wayne A. Wymore. *The Tricotyledon Theory of System Design*, pages 119–132. Springer-Verlag, New York, 1984.

[28] Wayne A. Wymore. *Model-based Systems Engineering*. CRC Press, 1993.

[29] Bernard P. Zeigler. *Multi-faceted modeling and discrete event simulation*. Academic Press, 1976.

[30] Bernard P. Zeigler. *Theory of Modeling and Simulation*. Interscience, 1976.

[31] Bernard P. Zeigler, Dale Fulton, Phil Hammonds, and Jim Nutaro. Framework for M&S – Based System Development and Testing in a Net-Centric Environment. *ITEA Journal of Test and Evaluation*, 26(3), 2005.

[32] Bernard P. Zeigler, A. Muzy, and E. Kofman. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Elsevier, 2018.

[33] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2 edition, 2000.

[34] Bernard P. Zeigler and Hessam S. Sarjoughian. *System Entity Structure Basics*, pages 27–37. Springer London, London, 2013.

[35] B.P. Zeigler and S. D. Chi. *Model-Based Architecture Concepts for Autonomous Systems Design and Simulation*, pages 57–78. Kluwer Academic Publisher, 1993.