Integration of Activity Specification into DEVS Modeling & Simulation Development Environment

Abdurrahman Alshareef[1], Bernard P. Zeigler[2]

[1]Information Systems Department – College of Computer and Information Sciences
King Saud University
P.O. Box 2454, Riyadh 11451, Saudi Arabia

[2]RTSync Corp.
6909 W Ray Rd STE 15-107, Chandler, AZ, USA

ashareef@ksu.edu.sa, zeigler@rtsync.com

Abstract. *We propose an integrative environment for the modeling and simulation of activity specification. The devised approach relies on the DEVS (Discrete Event System Specification) formalism for the foundational semantics of the essential activity elements. The code generation takes place afterward, targeting specific DEVS-compliant modeling and simulation (M&S) environments such as DEVS-Suite and MS4 Me. The modelers can set parameters or modify the code to satisfy specific needs. The simulation can then be conducted with behavior monitoring and visualization. We demonstrate the approach with observations about performance evaluation and tracking. Such environments have the potential to facilitate computational model development for System of Systems via full-scale simulation support.*

1. Introduction

There is a growing interest in developing environments that provide model execution and simulation for models specified at a higher level abstraction [Mohlin 2010, OMG 2018, MS4 Systems 2018, Eclipse Foundation 2019, NoMagic 2020]. Such abstractions tend to invoke and enrich the modeling of different scenarios and behaviors in various domains and perhaps cross domains. However, they can become challenging to realize in more concrete settings and environments. The simulation modeling can deliver more insightful observations of the system under study that is being modeled with such abstractions.

Among these abstractions are some languages adopted in software as well as system engineering problems. For example, the Unified Modeling Language (UML) and the System Modeling Language (SysML) are two popular cases where such an issue can exist, that is, the realization of the higher-level concepts in the lower level executions. Thus, there has been a growing interest in the last decade to equip them with model execution and simulation. The foundational semantics for executable UML subset (fUML) [OMG 2018] has been proposed and adopted in some recent tools development. However, some of the developed solutions for executable modeling have relied on fUML and the precise semantics of UML state machines (PSSM) [OMG 2019], which is an extension of the former. These two specifications, along with their execution engines, do not have an account of some aspects of the system, such as the temporal structure, since they primarily target the UML in general from an execution standpoint.

The model can be conceived in various forms, such as physical, mathematical, and logical representations. It can take the form of a visual, textual, or mathematical formulation. It can be written in a set-theoretic specification or according to the syntax of some programming language. From a simulation modeling standpoint, it is "a set of instructions, rules, equations, or constraints for generating I/O behavior" [Zeigler et al. 2018]. The challenge arises when such models are too abstract in a way that creates a major obstacle when someone tries to grasp a precise expression out of them or conduct a rigorous study or experiment. Nevertheless, this abstract way of deriving the models is a necessary part, or useful to say the least, of the modeling process at human thinking as well as the organization level. For example, interoperability emerges in conditions where multiple agents or constituents of some system have successfully identified their higher concepts that will most likely cause the desired impact at the operational level.

Therefore, the integrative approach to such issues can deliver some understanding of the mostly different models in place. Some models are abstract, with the possibility of running into multiple interpretations. Others have a large degree of precision, leaving no room for different adaptations. Putting these different models in one environment, which is the main contribution of this paper, may facilitate model development and reveal interesting insights or stimulate posing questions akin to the mental activity triggered in simulation and experimentation. The path toward such endeavor by no means should be overlooked. There are infinite possibilities, and navigating through them must take place with sufficient care.

In this paper, we discuss in details the model development steps in such an integrative environment for the modeling and simulation (M&S). It is organized as follows. The background section discusses our previously given definition of Activity and the DEVS formalism (Section 2). Then, we explain the overall modeling life-cycle in the proposed M&S environment (Section 3). In Section 4, we illustrate the life-cycle by going through the simulation modeling steps for an example *increment* operation. In Section 5, we go through the same steps but briefly in a more holistic manner for an example simulation for comparing the performance of single versus multiple servers.

2. Background

2.1. Activities

In this section we review our previous definition of Activity [Alshareef 2019]. We consider an essential set of activity elements. The Activity is primarily a graph that consists of nodes and edges that further specialize in many different nodes and edge types, mostly for nodes. An example of object nodes is the parameter that can refer to distribution and defines the arrival, or departure, of inputs, or outputs. Control nodes can be *merge*, *decision*, *join*, *fork*, *initial*, and *final* nodes, each of which describes a different type of control over the flow. The *merge* and *decision* nodes select at least one incoming or outgoing flows to proceed after evaluating its corresponding condition. We refer to both nodes as *Select* element. In SysML, the likelihood of traversing the outgoing flow of a decision node can be specified with probabilities. The *join* and *fork* nodes synchronize the incoming or outgoing flows waiting for all incoming flows in *join* and resulting in concurrent flows in *fork*. We refer to them as *Sync* elements. The *initial* and *final* indicates the initiation and termination of an activity. The other important type of node is the action node. It is the super-type of many specialized types of actions in the fUML model.

2.2. The DEVS formalism

The Parallel DEVS model [Zeigler et al. 2018] is $\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$. X is the set of input events. S is the set of sequential states. Y is the set of output events. δ_{int}, δ_{ext}, and δ_{con} are the internal, external, and confluent transition function, respectively. λ is the output function, and ta is the time advance function. A specification is devised for each type of activity node. Primarily, three atomic models are specified to correspond to *action*, *join* and *fork*, and *merge* and *decision*. These three atomic models mimic the behavior of their corresponding nodes during the simulation according to their ascribing semantics. Other aspects of the models are also captured, such as the flows, I/O parameters, and pins.

In a DEVS-based simulation of activities, the atomic model specification describes the action as well as the control node. We refer the reader to [Alshareef 2019] for further details about the DEVS formal specification of the action, and four control nodes that are *merge*, *decision*, *join*, and *fork*. The DEVS correspondent to the activity diagram is then the coupled model created based on the structure of the given Activity. Stimuli can be received by either a generator component or through an external input coupling resulting in the activation.

3. DEVS-based Support for Simulation Modeling Activities

The proposed environment relies on the DEVS as an underlying formalism for specifying the semantics of a set of activity elements. The identified set captures sufficient yet rich semantics associated with activities at a general level. The Parallel-DEVS, for example, provides a suitable candidate for the simulation of parallelism aspects and support offered by activities. The set consists mainly of *Action*, *Sync*, *Select*, *Parameter*, *Activity*, and *Flow* (see the palette in Figure 1). The *Action* represents a general way to capture many different specializations, such as processing, value specification, or object manipulation. These specializations are encountered in various system models and programming languages. The *Sync* is a general specification for fork and join nodes that are responsible for synchronizing the flow, whether it is incoming or outgoing. The *Select* is a general specification for decision and merging nodes that are responsible for selecting the incoming or outgoing flow. The *Parameter* can be input or output parameters to the Activity. The *Activity* can be placed within another activity in a hierarchical manner [Alshareef and Sarjoughian 2019] in order to set the stage for exploiting the support of modeling hierarchy offered in DEVS. Finally, the *Flow* is a general concept to represent different types of flow, such as object or control. Each *Action*, *Sync*, and *Select* map to an atomic models in the DEVS corresponding representation. The *Activity* maps to a coupled model. While the *Parameter* maps to an I/O port and the *Flow* maps to an external or internal coupling [Alshareef and Sarjoughian 2017].

3.1. The model lifecycle

The modeling starts by drawing the desired activity diagram using the palette. After doing so, the code generation can takes place. Currently, we have implemented three code generation facilities. The first one generates Java code for the simulation in DEVS-Suite. The time advance function is set with fixed time steps to provide the initial simulation. The second facility generates the code similarly but for the simulation in the MS4 Me environment. The third facility generates the code to be simulated in the same environment

however, with Markov time advance and state transition function. The resulting models are known as DEVS-Markov [Seo et al. 2018, Zeigler et al. 2018]. All of the three sets of classes are generated automatically.

Then, the modeler can interject by setting further parameters or modify the code to satisfy specific needs. For example, the time advance function can be set according to different timing assignment or distribution. The same can take place for all other functions, including their internal specification, to dictate certain behavior. The target M&S environment (i.e., MS4 Me or DEVS-Suite, as of now) can then conduct the simulation run along with the support for behavior monitoring and visualization offered in each tool, such as the simulation view [ACIMS 2019, MS4 Systems 2018] and super-dense time trajectories [ACIMS 2019]. The code generation can be extended to add further facilities to support the simulation in any environment that implements the DEVS abstract simulator.

The process can continue after observing the simulation and obtaining the results of it. The modeler can evaluate the performance and potentially use the resulting simulation to feedback the drawing of the activity diagram, setting the parameters, or making some changes to the code. Then, it goes through the simulation rerun, and so forth. Figure 1 highlights the important steps of the life-cycle in the proposed environment.

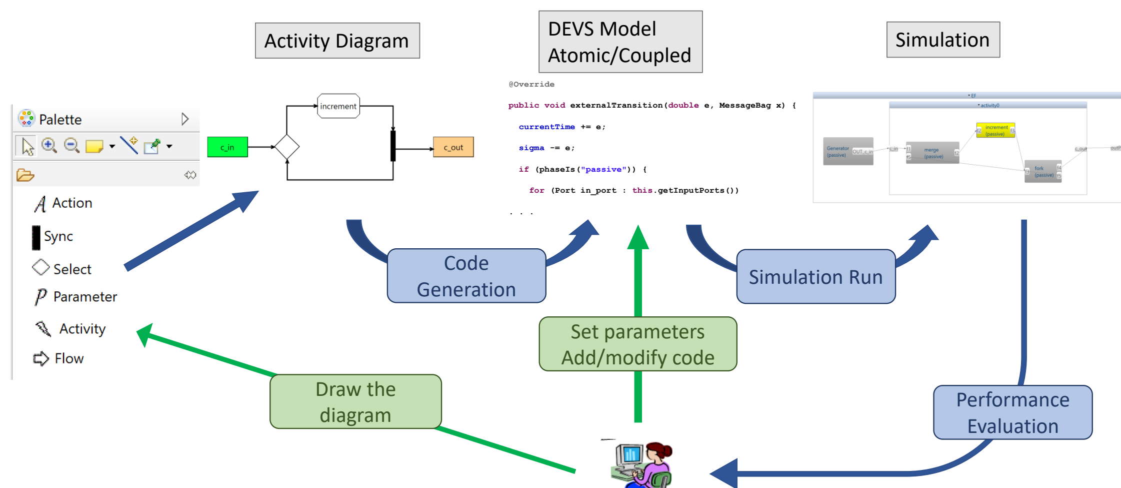

Figure 1. Integration of Activity Specification into DEVS M&S Development Environment

4. Simulation modeling *increment*

We will use this simple example to go through the model life-cycle steps with explanations. The *increment* is a basic operation to increase an integer by one. This example has been used in the tutorial provided by Moka execution engine [Eclipse Foundation 2019]. We use the same example to demonstrate the DEVS-based simulation support for operations or instructions that can take place at this level of development. In previous work [Alshareef et al. 2020], we discussed the example with details in both approaches with some comparison and evaluation remarks.

4.1. Drawing the activity diagram

The modeling initiates by drawing the activity diagram by selecting from the elements mentioned above the most suitable one to describe the desired behavior. The nodes can

be categorized into two groups. The first group is the action nodes that currently consists of the *Action* only and can be extended to describe more specific cases. The second group is the control nodes, which currently consists of the two elements *Sync* and *Select*. The *Sync* describes the flow where some form of synchronization is required. *Sync* with multiple incoming flow corresponds to the *join* node in activities. It is a maximum flow in a way that it requires to receive input through all of its incoming flows. The *Sync* with multiple outgoing flows produces its output to all of them simultaneously. Conversely, the *Select* is a minimum element by which only one flow is needed for proceeding. In the case of multiple incoming flows, it is a merging node where one input arrival is sufficient to trigger producing an output. While *Select* with multiple outgoing flows is a decision node by which some outgoing flow is selected to proceed where the condition is met.

The remaining elements, the *Parameter*, the *Activity*, and the *Flow*, allow for modeling I/O for the Activity, hierarchical activities, and the flow connecting these elements, respectively. The parameter can be an input for receiving stimuli, activation, or any distribution. It can also be an output to produce the result of the simulated behavior or some cycle thereof. The Activity can contain another activity in a hierarchical manner [Alshareef and Sarjoughian 2019]. In the activity diagram, such a notion is modeled using communication signals to either pause the flow and call some other behavior or operation or do the same thing without a pause. We have extended the Activity with this notion exploiting the hierarchical DEVS since we will be using it for the simulation modeling. The flow can be categorized into control or object flow. However, we only model it using this generic element since the distinction can be carried out using the I/O communicated between the different components during the simulation.

Modeling the *increment* operation can take various form. A simple activity in Figure 2 consists of an input parameter to receive the activation token. Then, a merging node allows any received flow to proceed in the outgoing flow. The first incoming flow, with the input parameter as a source, allows for the activation of this Activity. While the second incoming flow, with the fork node as a source, allows for the feedback loop after completing the *increment* action in order to initiate the following rounds. The *increment* action is responsible for conducting the actual operation. After that, it produces the output containing the new counter value after the operation. The fork node then simultaneously sends out the output to the output parameter and back to the merging node in order to continue the procedure. Figure 3 show a screenshot of the simulation after completing the third round. We note that this sort of scenario can be described with different diagrams. Moreover, further elaboration can also take place, such as the distinctions between different inputs. In [Alshareef et al. 2020], we discussed another scenario where a specific counter is associated with each different input or type thereof.

4.2. DEVS models code generation

After finishing drawing the diagram, the code generation process takes place. The generators are devised based on the proposed DEVS specification for the activity elements [Alshareef 2019]. The resulting codes are DEVS-compliant. Currently, the target simulators of the generation process are DEVS-Suite [ACIMS 2019] and MS4 Me [MS4 Systems 2018]. Since the drawn Activity in the previous step does not specify state explicitly, the specification describes that in a way that captures the essence of the activity-based semantics while the modelers are allowed to make changes as desired. The *Action*

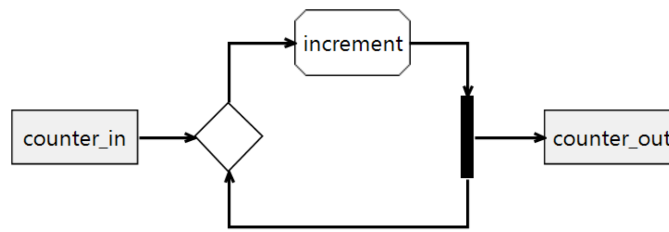

Figure 2. The activity abstraction for the *increment* example

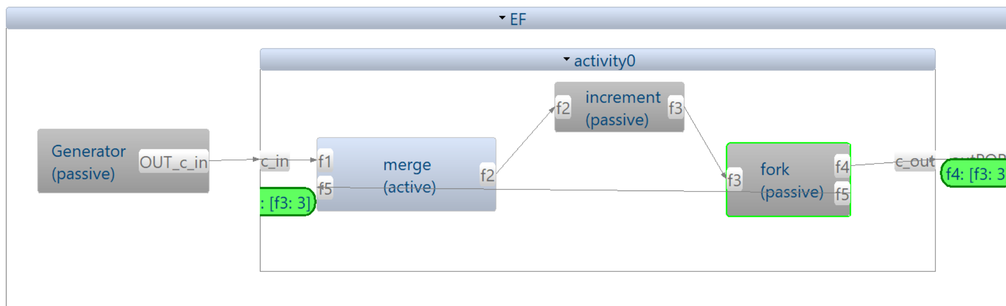

Figure 3. Simulation view after completing three rounds

initially has two phases that are *active* and *passive*. The *active*, or *busy*, state represents action being consumed in some task or, in other terms, having the token and executing. In contrast, the *passive* state represents the action being idle. The *Sync* has three different phases in order to capture the essence of the fork and join semantics. A *waiting* state is added to the phase set to allow for synchronizing incoming flow. However, the model does not transition to this state if it is a fork node since it does not expect other inputs. The *Select* state has a set of conditions as state variables to facilitate the determination of which outgoing flow is supposed to proceed.

Each *Action*, *Sync*, and *Select* will have a corresponding class generated in Java since it is the language for the target simulators. The class extends the atomic model class. The *Activity* will have a class as well; however, it extends the coupled model properties and features since the Activity corresponds to a coupled model in the proposed mapping. The *Parameter* will correspond to ports in the generated classes, and the couplings will be determined based on the flows.

The time advance function is set in various ways in the initial classes and can be set further by the modelers. We currently generate three sets of classes. The first set consists of Java classes with fixed time assignments for simulation in DEVS-Suite. The second set is the same however, for the simulation in MS4 Me [MS4 Systems 2018]. While the third set consists of classes that correspond to DEVS-Markov models, which are stochastic models with Markovian state transition and time advance.

4.3. Setting parameters and making code changes

The proposed environment takes an abstraction view that is complementary to code. Since modeling involves lower-level equations and operations, the generated DEVS model in the code format can undertake further modification and enhancement, especially concerning the operational level and parameter setting. Such modifications might be convenient

to be encoded directly into the concrete model in a consistent manner without losing the prior specification of the abstraction. We note that this step is optional, and the automatically generated code can proceed directly for the simulation step without modifications.

In this example, the basic operation that is taking place is the *increment*. It is simple enough, and its syntactical representation is very close to its abstract one. Therefore, it does not require substantial modeling effort with logic and procedure. The modelers do not need to have advanced knowledge of programming techniques and data structures in order to encode such an operation. Therefore, we delegate this step to the concrete code to preserve a more succinct activity diagram that focuses on the logic and semantics of the model. Figure 4 shows a code snippet for MS4 Me corresponding to the external transition function in the atomic model for the *increment* action. The tool generates this code automatically with an initial and default value for the job, while the change to represent the desired operation is highlighted and made on the generated code.

```
@Override
public void externalTransition(double e, MessageBag x) {
  currentTime += e;
  sigma -= e;
  if (phaseIs("passive")) {
    for (Port in_port : this.getInputPorts())
      if (x.hasMessages(in_port)) {
        ArrayList<Message<Serializable>> messageList = in_port.getMessages(x);
        for (int i = 0; i < messageList.size(); i++) {
          job = counter+++"";
          holdIn("active", processing_time);
        }
      }
  }
}
```

Manually modified after the code generation step

Figure 4. The code snippet for MS4 Me corresponding to the external transition function in *increment* action

4.4. Running the simulation

The code for the DEVS models is now ready for conducting the simulation. As part of the code generation process, a generator is added to the model to initiate the simulation with inputs. Various generators can be used for this purpose using various uniform or statistical distributions. The simulation view in MS4 Me supports running the simulation at once. It also allows stepping through each iteration, running a specific number of iterations, or to the desired simulation time. It shows the modelers the simulation status and some animations of the dynamics such as I/O communication amongst the components. The simulation can be evaluated through such means or via designing an experimental frame, especially for models with some degree of complexity and scale. In DEVS-Suite, different types of time trajectory, including super-dense time, can be generated concerning state variables or I/O, which can be useful in many cases for observing the logic of the created model or evaluating their performance. In previous work [Alshareef and Sarjoughian 2018], we showed some cases of such examination and highlighted its importance, especially in the case of modeling activities.

In this *increment* example, only one input needs to feed into the Activity for the activation. After that, the model will loop around the action due to having a feedback flow outgoing from the fork node. There could be, however, other scenarios. For example, there could be another input from the generator for deactivation.

4.5. Evaluating the performance

Observing the simulation run, whether via the offered visualization and animation features or some designed experimental frame, is significant for performance evaluation of the model. In previous works [Alshareef and Sarjoughian 2018, Alshareef 2019], we have examined some measures of evaluation in multi-processing architectures such as throughput and parallelism. The same measures can apply to the *increment* example or other models to examine specific aspects of their logic. Moreover, models can be subjected to further constraints, such as timings, for more advanced analysis as opposed to basic debugging. We devise an activity abstraction to correspond to various multi-processing schemes. Then we examine intrinsic characteristics of these schemes using the generated code for the DEVS models to verify and compare. Parallel execution was an essential aspect of particular interest to examine parallel flow in the activity specification. An engine with support for parallel simulation is necessary to underline this aspect.

5. Single server versus multiple server example

The example consists of a server or multiple servers that receive jobs for processing and could be linked together in different archetype architectures. The example is presented initially in the Simulink SimEvents documentation center [Mathworks 2018] to compare between different server architectures considering different service times. The experiment aims to conclude that one fast server is better than multiple slower servers when it comes to the average waiting time. The architecture is designed with two settings. The one server model has the required service time for each job generated based on exponential distribution with mean $\mu = 1$. The second case has three servers with mean $\mu = 3$ for generating service time periods. In both cases, the arrival of jobs is generated based on exponential distribution with $\mu = 2$. The experiment concludes that the turnaround time achieved by the single but fast server is less than the turnaround time achieved by the multiple but slower servers.

The server systems are described by [Wymore 1993] with servers being developed after examination of their counterparts in a manufacturing system model. Two models are developed using the activity specification from a system standpoint. Figure 5 shows the Activity corresponding to the multiple servers architecture. The single server architecture has one action only instead. The *merge* node will be equipped with a queue at the DEVS-level to hold on jobs in case none of the servers is available at the moment of the job arrival. The calculation of the turnaround time starts recording until the job processing is completed. We note that turnaround time is the total time for the job being in the system and includes the waiting time to be served as well as the time during the service ([Mathworks 2018] refers to this quantity as the waiting time.) The notification of the availability of the server is sent back to the *merge* node via feedback flow outgoing from the fork node. The full details of the experiment are captured at the DEVS level.

The resulting DEVS models are evaluated using an appropriate experimental frame to confirm the aforementioned conclusion. The time required for processing each

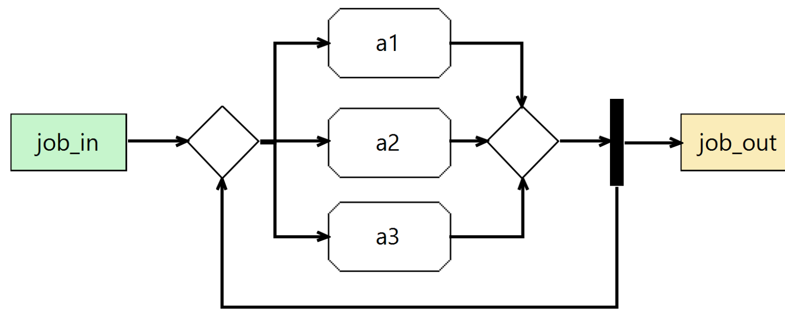

Figure 5. An activity diagram for the multi-server system with three actions

job is set with μ equal to the number of servers while μ for job arrival is 2 for all runs. The results on turnaround time are shown in Table 1, where a different number of servers is specified and simulated. The table shows average turnaround times for service of 100 jobs for the cases of 1, 3, 5, and 6 servers.

Number of servers	1	3	5	6
$Turnaround\ time$	1.241	2.659	4.378	5.254

Table 1. The result of simulation with different turnaround time for different number of servers and μ equal to the number of servers in each case for the job service time

The example briefly highlights our approach and demonstrates how analysis with some degree of rigor can take place within activities that are developed to describe flow in the system or some parts thereof. The experimental frame for turnaround time can be generalized to correspond to many aspects of the design of software architecture. In addition to the numerical analysis, the use of M&S can provide valuable insights regarding specific procedures or steps thereof with varying degrees of temporal precision or resolution. The DEVS Markov mapping also shows how the approach can account for stochastic modeling which can be useful in predicting and enhancing the analysis of different alternatives. Whether a system is under study or development, simulation models can continue to be developed and become part of the actual running system in a seamless manner.

6. Future work

Software errors become far more threatening when they are within a software-intensive system or cyber-physical system environments due to their prevailing impact on the physical world. It is crucial to treat parts of these systems with more rigorous approaches, especially the ones with safety implications in critical systems. It can be helpful to enable the modeling and simulation of these parts at different stages of the analysis and design.

We plan to continue examining further techniques for the proposed integrative environment to enhance the model life-cycle. In particular, we are examining different ways to incorporate the abstraction concepts along with the lower-level components of the model in a systematic manner. The model in this approach currently can be further refined to the hardware-level design, although in an ad-hoc way. We are currently working to demonstrate its applicability and use in domains with varying levels of complexity, such as Cyber-Physical Systems, Internet of Things, and cloud-based systems.

References

ACIMS (2019). DEVS-Suite Simulator version 5.0.0. Available at `https://sourceforge.net/projects/devs-suitesim/` (Accessed July 1, 2020).

Alshareef, A. (2019). Activity specification for time-based discrete event simulation models. *Ph.D. Dissertation, Arizona State University*.

Alshareef, A., Kim, D., Seo, C., and Zeigler, B. P. (2020). Activity Diagrams between DEVS-based modeling & simulation and fUML-based model execution. In *Proceedings of the 2020 Summer Simulation Conference*. Society for Computer Simulation International.

Alshareef, A. and Sarjoughian, H. S. (2017). DEVS specification for modeling and simulation of the UML activities. In *Proceedings of the Symposium on Model-driven Approaches for Simulation Engineering*.

Alshareef, A. and Sarjoughian, H. S. (2018). Parallelism semantics in modeling activities. In *Proceedings of the Theory of Modeling and Simulation Symposium, SpringSim (TMS) 2018, Baltimore, MD, USA, April 15-18, 2018*, pages 6:1–6:12.

Alshareef, A. and Sarjoughian, H. S. (2019). Metamodeling activities for hierarchical component-based models. In *Proceedings of SpringSim, Theory and Foundations of Modeling & Simulation (TMS), Tucson, Arizona, USA*. Society for Computer Simulation International.

Eclipse Foundation (2019). Moka release (4.0.0) for Eclipse Papyrus release (4.7.0). Available at `https://eclipse.org/papyrus/` (Accessed July 1, 2020).

Mathworks (2018). Simulink. Available at `https://www.mathworks.com/products/simulink.html`.

Mohlin, M. (2010). Model simulation in rational software architect: simulating uml models. *Cupertino, CA: IBM*.

MS4 Systems (2018). MS4 Me Simulator version 3.0. Available at `http://ms4systems.com/pages/ms4me.php` (Accessed July 1, 2020).

NoMagic (2020). Cameo Systems Modeler. Available at `https://www.nomagic.com/products/cameo-systems-modeler` (Accessed July 1, 2020).

OMG (2018). Semantics of a Foundational Subset for Executable UML Models (fUML) version 1.4.

OMG (2019). Precise Semantics of UML State Machines version 1.0.

Seo, C., Zeigler, B. P., and Kim, D. (2018). DEVS Markov modeling and simulation: formal definition and implementation. In *Proceedings of the 4th ACM Inter. Conference of for Engineering and Sciences*.

Wymore, A. W. (1993). *Model-based systems engineering*, volume 3. CRC press.

Zeigler, B. P., Muzy, A., and Kofman, E. (2018). *Theory of modeling and simulation: discrete event and iterative system computational foundations*. Academic press.