IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 1, 2021, accepted May 21, 2021, date of publication May 31, 2021, date of current version June 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3084940

Hierarchical Activity-Based Models for Control Flows in Parallel Discrete Event System Specification Simulation Models

ABDURRAHMAN ALSHAREEF[1] AND HESSAM S. SARJOUGHIAN[2]

[1]Information Systems Department, College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia
[2]Arizona Center for Integrative Modeling and Simulation, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281, USA

Corresponding author: Abdurrahman Alshareef (ashareef@ksu.edu.sa)

ABSTRACT Behavior modeling grounded in the Discrete-Event System Specification (DEVS) and Unified Modeling Language (UML) activity specifications is crucial for simulating dynamical systems. The Model-Driven Architecture (MDA) design approach provides flexible yet rigorous layered metamodels for the UML activity diagrams. Our approach for behavior modeling is focused on the action and control concepts in the UML activity metamodels and realizing them as artifacts according to the DEVS formalism. The syntax and semantics for the artifacts conform to the parallel DEVS model specification and execution protocol. We use the system-theoretic state, component, and hierarchy concepts as the foundation for formulating the DEVS Activity models and supported with a prototype graphical tool developed in Sirius. This research also proposes the Parallel DEVS as a formal approach for examining the semantics of the UML Activities. We develop, simulate, and analyze a set of prototypical multi-processor architecture systems demonstrating different synchronization and selection schemes using the DEVS-Suite and MS4 Me simulators.

INDEX TERMS Activity diagrams, behavior modeling, DEVS, parallelism, model-based design, modeling & simulation, software modeling.

I. INTRODUCTION

Dealing with different parts of a system model can be problematic, particularly when the parts are scattered within and across different abstraction hierarchies. At some point, the abstraction layers of a hierarchy, each possibly having multiple levels within, have to be bounded with some constraints to make them useful and prevent issues such as circularity relationships. For example, the model-driven architecture (MDA) is defined as a four-layer hierarchy from M0 through M3, where the former represents the most concrete, and the latter represents the most abstract. Although useful, determining the boundaries and relationships among the layers in a clear-cut manner, particularly from the standpoint of executable models and simulation, is difficult. Navigating through many possible relationships across and between the abstraction layers using different kinds of extension and instantiation schemes poses further challenges for the simulation models.

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaogang Jin.

The problem of partitioning models into components and relationships becomes evident for structure as well as behavior. For the structure, complications may arise with a significant increase of multiple message types and communications requiring computation synchronization and concurrency. For the behavior, dissecting the internals of communicating components of a system can also pose difficulties in developing executions for the abstraction. At some point, nonetheless, behavioral specifications at multiple levels of abstraction must take place in controlling and conducting some lower-level computing tasks. The delegation of lower-level tasks becomes challenging due to the central role abstraction hierarchies play in managing complexity and scale across complementary model specifications. Having a well-defined formal model specification for control flow nodes helps in providing a concise basis to understand and analyze their behaviors.

Considering the ongoing efforts in further deepening the hierarchy for component-based models, we propose examining and using the action and control elements at the meta-layer activities, mainly focused on the M1 and

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

M2 layers. The system-theoretic state, component, and hierarchy concepts are used in defining the action and control elements essential for specifying model behavior. One of our goals is to understand the different roles played by different layers encompassing DEVS and UML, especially from multiple meta-layers viewpoints. The specifications of such layers are distinct and explicit for simulation modeling purposes.

In discrete systems [1]–[3], a variety of needs institute the time scale upon which the system representation or approximation can execute. Real systems such as smart manufacturing or transportation are known to be large in both scale and complexity. They have numerous components with various types of connections among them. Components do not have to resemble each other except at high levels of abstraction that are often rendered difficult to concretize in the computational models. Given these types of systems, some models, such as synchronous reactive components, provide strong constraints for timing, state change, and composition. The specification of synchronization nodes in DEVS allows for the concrete realizations to take place in a disciplined manner.

Discrete models of discrete systems can have asynchronous behaviors. For a complex discrete system, simpler models with single-input and single-output can be placed within models that result in different input and output multiplicities and behaviors. Observing the degrees of detail with sufficient confidence is challenging yet essential to develop, simulate, and analyze models. Interesting results can disperse via simulation studies; however, sophisticated observations can be accessible and clearly understood only through rigorous experiments. Useful analyses such as throughput, as we will demonstrate in section VII, might be achievable through making such models subject to experimental designs. Still, such analyses with a keen sensitivity to temporal aspects can be intractable or otherwise impractical to carry out for arbitrary simulations.

The contribution of this paper is the grounding of our previous works on a behavioral modeling framework supported with the MDA metamodeling, UML activity diagrams, and parallel DEVS formalism. This framework lays out the basis for an activity-based modeling approach focusing on action and control nodes in a flow-based manner for simulation modeling and, in particular, for parallel DEVS models. We examine activity specification as a standalone approach across different modeling meta-layers to develop useful simulatable models. We first discuss the MDA architecture and DEVS modeling frameworks. Then, we present the DEVS specification for activity modeling. In the remaining sections, we detail the semantics of the modeling approach in conjunction with demonstrations of certain aspects of multi-processing architectures.

II. SIMULATION MODELING ARCHITECTURES AND FRAMEWORKS

We begin with discussions about related works and background regarding the development of architectures and frameworks to support discrete event simulation modeling. First, we briefly describe MDA and then discuss some of its concepts, particularly when applied to modeling and simulation. Second, we highlight a few existing studies and present the researchers' viewpoints regarding what accounts for the ongoing efforts in dealing with models that can be developed using different abstraction means.

A. MODELING LAYERS

In an earlier study [4], we proposed a metamodel for the DEVS atomic model spanning the MDA concepts and techniques. We extend the core Eclipse Modeling Framework (EMF) [5] model with primitive notions for behavioral specifications to make behavioral modeling possible along with structural specifications. Although useful, there are some inherent limitations of using such means for behavioral modeling.

The MDA layers M3, M2, M1, and M0 lay the groundwork and guidelines for incrementally developing models of component-based systems. The guidelines are useful if followed carefully. However, the nature of extension techniques among metamodels may result in unintended and unnecessary complexity, and overhead [6]. Concepts at a meta-layer necessitate further efforts to substantiate them at the next lower meta-layer, often demanding a significant effort. The key idea is to create a classifier and multiple extensions and instances thereof across all the MDA layers. The directions of extension and instances are thought to be orthogonal. However, the Object Management Group (OMG) has made more elaborate standards by which distinctions between cross meta-layers and within a single meta-layer are drawn. Interpretation and instantiation ascribe the definitions in meta-layers.

In some cases, extensions reside horizontally within the same meta-layer, and instances reside vertically in the next lower layer. The connection is reversed in other cases. In our work, we observe both standards and deal with such complications by relying on the theory of modeling and simulation, modular, hierarchical DEVS in particular, for the demonstration from a system-theoretic standpoint.

It is essential to establish a more rigorous means of facilitating the creation of models at a concrete layer. It is also significant and yet far more challenging to realize, with mathematical rigor, connections between models at the concrete layer and their counterpart abstractions at some high layer. The difficulties symmetrically increase with the layers that are higher in the hierarchy. We examine the concepts and present works that attempted to map concepts from upper layers downward. The results are promising for relatively simple systems but not as much for complex systems. We will discuss this further in the following section.

B. RELATED WORK

Leaping models and abstractions of models to some counterpart manifestation at a concrete layer is increasingly recognized as a challenging problem. Some researchers have focused primarily on efforts to realize implementations of

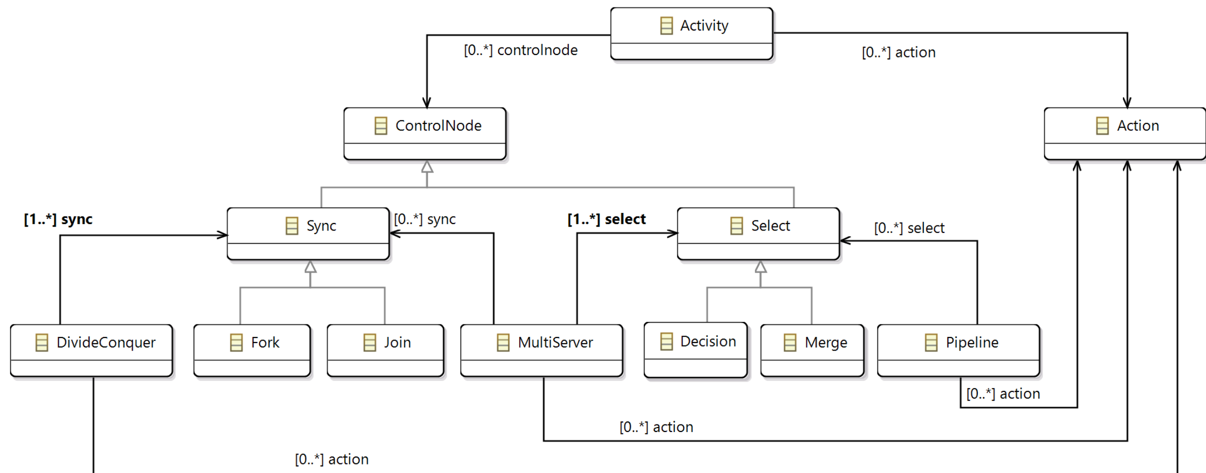


FIGURE 1. A metamodel at the M2 layer for modeling activities with associations to the considered multi-processing regimes.

models based on model-driven and model-based system engineering (MBSE) known as MBSE/M&S in the sequel [7]. In statecharts [8], models are perceived to construct the system under development. In some others, the system viewpoint is absent, and yet the effort is focused on ascribing semantics for the intended modeling language [9]. While many others make a more deliberate attempt toward employment of model-driven frameworks to integrate simulation as a means for precisely observing the system under study [10]–[13].

Efforts have been made to utilize MDA to provide model transformation frameworks. A general objective in certain studies [10], [11] was to promote model reuse across different platforms. It is quite often the case that specific capabilities are offered in a target platform but do not apply in others, which led to the notion of platform-independent solutions. The problem may also become more difficult for a number of considerations. Different abstractions of time pose key challenges across execution environments. Transforming models with time agnostic or implicit timing to those that require explicit timing is an example. Timing is a significant issue in transforming UML and DEVS models and their modeling environments. MDA alone falls short of providing a concrete solution for behavioral specification, although it has been essential in guiding certain advances in different DEVS-based modeling frameworks (e.g., [10], [11], [14]).

On the one hand, efforts and standards have pushed toward enabling the creation of a platform-independent model [15], [16]. On the other hand, the feasibility of executing these models with techniques such as code generation is subject to fundamental limitations with knowledge gaps and arbitrary semantics [17]–[19]. Simulations of complex systems are of particular importance when it comes to realizing behavioral specifications. As such, this paper offers a step toward achieving this goal.

In previous works [19]–[21], we laid a groundwork for the modeling and simulation of activities using the parallel DEVS formalism with its abstract simulator. Some exemplary models demonstrate a basic mapping from activity action to the atomic DEVS model specification [20]. The mapping attempts to utilize a rich simulation framework in addition to debugging [22] or execution with a fixed time step. We discussed the approach and the mapping in more detail about the I/O function in the system specification hierarchy [21]. More recently [23], we extended the work to the coupled component with a focus on the model hierarchy to facilitate the construction of the component-based models.

III. ARCHITECTURES FOR MULTI-LAYER MODELING ACTIVITIES

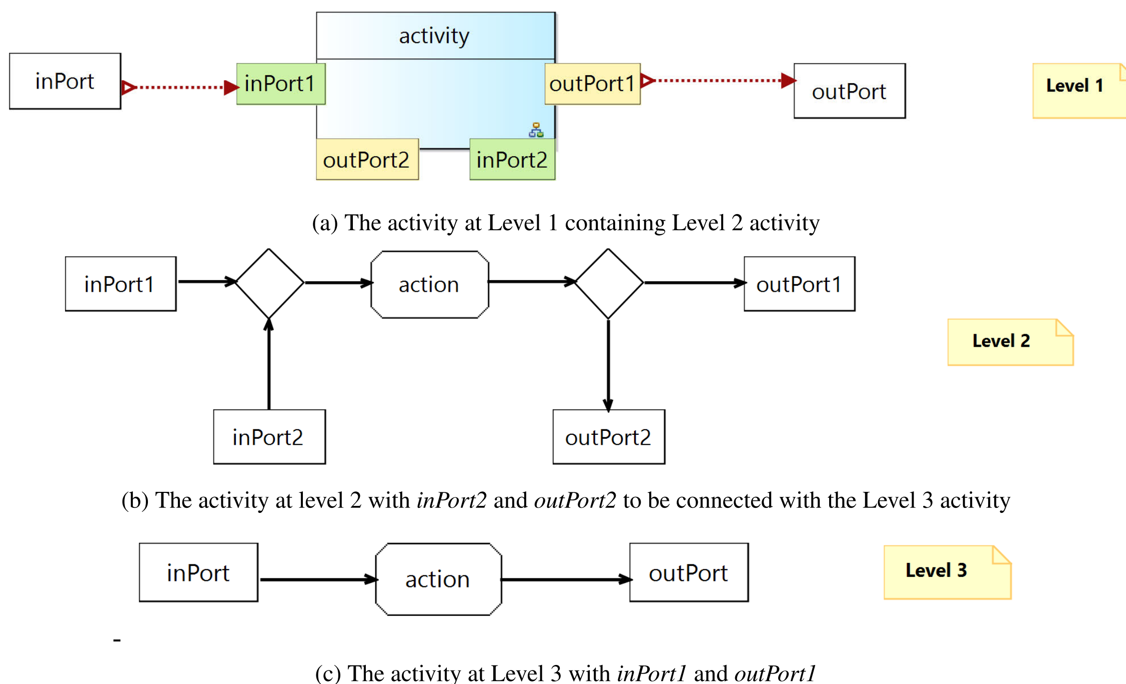
We propose taking two different views in creating multiple layers for modeling activities. The first is to consider MDA layers and place every model and metamodel within them. The second is to account for a free form layering system and place activity models and metamodels within their respective levels according to a given domain-specific model. Next, we discuss these views and provide a brief comparison with the advantages and disadvantages for each.

A. CONFORMANCE TO THE MDA LAYERS

In this perspective, we confine all models and metamodels to MDA. The Meta-Object Facility (MOF) at M3 and the activity metamodel with our proposed notions at M2 with horizontal extension techniques are used to define the abstract concepts for the considered multi-processing regimes (see Figure 1). Then, the instantiations, or extensions at the lower layer according to the other perspective, are made at M1 such as the ones shown in Figure 5 and 7, which we will discuss with more details in section V and VI.

B. CONFORMANCE TO THE DEVS FORMALISM

In this perspective, we propose lifting conformance to the MDA and instead use domain-specific knowledge to define activities in layers forming a well-formed hierarchical structure. This view offers a wide range of possibilities while benefiting from layered specifications in general without



(a) The activity at Level 1 containing Level 2 activity

(b) The activity at level 2 with *inPort2* and *outPort2* to be connected with the Level 3 activity

(c) The activity at Level 3 with *inPort1* and *outPort1*

FIGURE 2. The multi-level modeling activities.

having, at least initially, to comply with the MDA layers. The MDA architectural benefits can be readily undermined when many extensions, whether vertically and/or horizontally, are needed. Furthermore, arriving at an early simulation is important from an iterative and incremental model development standpoint. This need can be subjected to a significant delay when conformance to the MDA standard is to be satisfied first. Such a delay can adversely impact the development of simulation studies invaluable for gaining early insights into requirements. Compliance with the four-layer architecture can take place at a later stage while benefiting from the DEVS formalism. The model layers account for basic composition and inheritance relationships while benefiting from MDA guidelines. The DEVS abstraction layers can be transformed into those of the MDA. We proposed grounding such architecture on the hierarchical and modular DEVS in previous work [23]. We note that the activity diagrams in Figure 2 have been devised using our tool that we have developed to visually create and then generate code for the open-source, free DEVS-Suite and the proprietary, commercial MS4 Me simulators.

To create a richer set of models, we propose using both architectures simultaneously as well as distinctly to benefit from the hierarchical elements defined in both MDA and DEVS. We will demonstrate methodically with model examples and simulations. Using both architectures may open up a wide range of design choices by which various alternative models can be enriched and explored to benefit the modeling process with some specific measures or at large. We will use them as a ground for the models created throughout the paper.

IV. DEVS SPECIFICATIONS FOR THE ACTION AND CONTROL NODES

Previous studies [19], [21] have examined different semantics of activities and created a set of specifications that correspond to various elements in the UML activity metamodel. We formulate the specifications primarily for two types of activity elements known as action nodes and control nodes.

In a nutshell, every activity is essentially a graph that consists of nodes and edges. Edges are referred to as flows, while the nodes can be an object, control, or action. Control nodes include *join*, *fork*, *merge*, and *decision*. The *fork* node is the one that synchronizes the production of outputs through its outgoing flows. Similarly, *join* synchronizes the flows but regarding its inputs where it expects an input through each incoming flow. Because they are symmetric, we will later refer to *fork* and *join* together as the *SYNC* specification (Listing 1). In the same vein, the *merge* and *decision* nodes are used to select one flow for proceeding. In the former, it is incoming, and in the latter, it is outgoing. We will refer to them jointly in the *SELECT* specification (Listing 2).

Action nodes are the most fundamental unit of behavior in the Unified Modeling Language (UML) 2.5 metamodel [16]. They are defined as an abstract node in the metamodel and are refined in the foundational subset of the executable UML Models (fUML) [24]. Sets of specific actions are sub-types of the abstract action. For example, one category suggests a collection of actions to be reading actions, and therefore, their specifications are partially defined in the UML specification. To provide an implementation for such descriptions, the standard provides a mapping to the Java programming language through interpretations. The capability of running fUML

models is delivered through model execution environments such as Moka within Papyrus [25]. The modeling capability is in Papyrus. In our work, we focus on providing an execution capability by exploiting a DEVS-compliant simulator such as the DEVS-Suite simulator [26]. We argue that the inherent account of time (i.e., as a standalone part of an executable model) is necessary, with varying degree, to navigate through the semantics of various activity constructs [19].

Control nodes in activities are also essential for guiding the flow. Their roles vary; however, we mainly categorize them into two major types. The first type consists of the *fork* and *join* nodes. The second type consists of the *decision* and *merge* nodes. Similarly, with what we just mentioned earlier, we capture the specification of the first type in the *SYNC* model and the second one in the *SELECT* model. These two major types mainly differ from each other in the synchronization of their incoming and outgoing flows. In the former type, the flows are synchronized, but that is not necessary for the latter. The specifications of different nodes are discussed in previous studies by [19]–[21].

Next, we define a formalized mathematical specification for each type according to the parallel DEVS formalism. Listing 1 shows a formal specification for the first type. The syntax and semantics of this specification, as with the second type, strictly conform to the parallel atomic DEVS model abstraction.

$$SYNC = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle, \text{ where}$$

$X^b = \{(p, v) : p \in IPorts, \ \ v \in X_p,$
$\quad IPorts = \{in_1, ..., in_n\}, \ X_p = V(\text{an arbitrary set})\}$
$\quad$ is the set of input port and value pairs
$Y^b = \{(p, v) : p \in OPorts, \ \ v \in Y_p,$
$\quad OPorts = \{out_1, ..., out_m\}, \ Y_p = V(\text{an arbitrary set})\}$
$\quad$ is the set of output port and value pairs
$S = phase \times \sigma \times task \times C, \text{ where}$
$\quad phase = \{passive, waiting, active\},$
$\sigma = \mathbb{R}^{+}_{0,\infty}, \ task \subseteq X^b,$
$\quad C = \{(p, c) : p \in IPorts, \ \ c \in \{true, false\}\}$
$\quad\quad$ is the set of input ports and conditions
$\delta_{ext}((phase, \sigma, C, task), e, X^b) =$
$\quad (waiting, \infty, (p_i, true), (p_i, v_i))$
$\quad\quad if \ p_i = in_i \ \wedge \exists (p_j, c) \wedge c = false \wedge i \neq j$
$\quad (active, 0, C, (p_i, v_i))$
$\quad\quad if \ p_i = in_i \ \wedge \forall (p_j, c) \wedge i \neq j \implies c = true$
$\delta_{int}(phase, \sigma, C, task) =$
$\quad (passive, \infty, (p_i, false), x) \ \ x \in X^b$
$\quad\quad if$ the queue is empty
$\quad (passive, \infty, (p_i, true), x) \ \ x \in X^b$
$\quad\quad if$ the queue is not empty
$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$
$\lambda(active, \sigma, task) = (p, task)$
$ta(phase, \sigma) = \sigma.$

LISTING 1. ***SYNC* atomic DEVS model specification.**

In the case of *join*, the node expects the arrival of input via all incoming flows before dispatching output. Therefore, this is represented in the *SYNC* specification by having multiple input ports. The state is used to distinguish incoming inputs arriving on multiple ports from one another. The distinction is carried out via $(p, c) \in C$. As soon as all expected inputs have arrived, the output is dispatched with a zero time advance assuming no delay is expected to take place for producing and dispatching the output. The *fork* behaves similarly, except that multiple outputs follow for a given input.

The formal specification corresponding the second type of control node (for *merge* and *decision*) is detailed in Listing 2.

$$SELECT = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle, \text{ where}$$

$X^b = \{(p, v) : p \in IPorts, \ \ v \in X_p,$
$\quad IPorts = \{in_1, ..., in_n\}, \ X_p = V(\text{an arbitrary set})\}$
$\quad$ is the set of input ports and values
$Y^b = \{(p, v) : p \in OPorts, \ \ v \in Y_p,$
$\quad OPorts = \{out_1, ..., out_m\}, \ Y_p = V(\text{an arbitrary set})\}$
$\quad$ is the set of output ports and values
$S = phase \times \sigma \times task \times C, \text{ where}$
$\quad phase = \{passive, sending\}, \ \sigma = \mathbb{R}^{+}_{0,\infty}, \ task \subseteq X^b$
$\quad C = \{(p, c) : p \in OPorts \ and \ c \in \{true, false\}\}$
$\quad\quad$ is the set of output ports and conditions
$\delta_{ext}((phase, \sigma, C, task), e, X^b) =$
$\quad (sending, 0, C, (in, v_1), \cdots, (in, v_n))$
$\delta_{int}(phase, \sigma, C, task) = (passive, \infty, (p, false), task)$
$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$
$\lambda(sending, \sigma, C, task) =$
$\quad (p_i, task) \ if \ (p_i, true)$
$ta(phase, \sigma) = \sigma.$

LISTING 2. ***SELECT* atomic DEVS model specification.**

The above specification is generalized for the *decision* and *merge* nodes. We note that elaboration is required to account for specific behavior types such as the DEVS Markov models. The structural part of this specification (input, output, and state constructs) is the same for the *SYNC* model. The behavioral specification represents the dynamics of the *SELECT* model. The *SYNC* and *SELECT* models have the same time advance function specification.

In previous work [20], we discussed in detail a discipline for a network switch example. The selection specification describes the *decision* node where the flow is directed in such a network based on a polarity condition. The condition is maintained and updated at each input arrival time. The *SELECT* specification resembles multiple aspects of the example in the previous work. We will discuss this specification with a slightly more concrete example in the following sections.

A. MAPPING THE UML ACTIVITY CONTROL, OBJECT, AND FLOW TO THE DEVS MODEL, PORT, AND COUPLING

We note that according to the activity metamodel in the UML, the notion of a pin can be defined only for the executable

activity node. The pin can be an input, an output, or a value pin. Other types of activity nodes (e.g., control node) cannot be defined with pins, and therefore, the handling of I/O is tacit or left undefined. In the DEVS formalism, an atomic model can be equipped with a finite but unrestricted number of ports. Whether it is input or output, each port can be arbitrarily attached to one or more internal or external coupling. Thus, we create two ports for each channel (i.e., coupling), one as an output and the other as an input, where the coupling is added to link the two ports. A channel, with its input and output ports, corresponds to a flow in the activity diagram. It serves as a means to transfer I/O through different models, whether they correspond to actions or control nodes. By assigning a distinct port to each coupling, we eliminate the possibility of duplicating I/O in the DEVS network and therefore needing some elaborate mechanism for handling many ports per coupling. Each I/O or some part thereof gets transferred only to the intended element.

From the UML vantage point, flow is classified by control and object flow types, each having its syntax and semantics. A closer examination of the flow types for the fUML model reveals the notion of *locus* to facilitate execution by carrying out (i.e., transmitting) information through activity nodes during the execution life cycle. Control flow is defined to dictate order, while object flow is defined to also dictate order but to do so while carrying data between different nodes. Control nodes do not distinguish between flow types, nor do they require pins to link with object flows. The pins are designated for carrying whichever kind of flow they are connected with. They pass any received object along to its designated nodes without any manipulation. An action can have, at most, one input pin and, at most, one output pin. An action node's input and output pins can be linked to object flows; other flows for the action node can be of a control type. Action can receive and produce as many object and control flows as needed, but a finite number of flows can link to action, whether incoming or outgoing. Control nodes cannot connect with pins, but they can relate with as many flows as needed, whether object or control. In our proposed approach, we ignore this classification, and every flow is defined as coupling. That is, we make no distinction between object and control flows. We account for the syntax mentioned above and semantics through the definitions of the port of atomic/coupled DEVS models. The locus is accounted for in both the specification as well as the simulation protocol while maintaining the separation between the model and the simulator. In addition, the arbitrary handling of I/O is due to the DEVS abstract execution protocol, which is domain agnostic.

The following subsections describe the mapping for three major activity components to their DEVS counterparts. First, we describe the mapping from *fork*/*join* activity nodes to *SYNC* atomic model. Second, we describe the mapping from *decision*/*merge* activity nodes to *SELECT* atomic model. Then, the activity as a whole is mapped to a coupled model in which all the corresponding atomic models are contained in it.

1) SYNC ACTIVITY MODEL

In this atomic model (Listing 1), one input port is defined as corresponding to each incoming flow. An output port is also designated for each outgoing flow. For example, if the *join* node has two incoming flows and one outgoing flow, then the correspondent atomic model would have two input ports and one output port. Then, a coupling is attached to each port. Therefore, the *SYNC* atomic model initializes in a passive state. As soon as it receives input through one of its input ports, it transitions to a waiting state until other incoming inputs arrive from different ports, and the output does not dispatch until all required inputs arrive. When an input arrives through each input port, then the model transitions to a different state, after which it combines all inputs and prepares the resulting outputs. The combining procedure is absent from the metamodel of activities (i.e., the specifics of a procedure are to reside in a concrete model according to some given application domain while having a default procedure in place for the initial simulation). The output then dispatches through all output ports, and therefore, their distinctive couplings are used for delivery to their destinations.

The *SYNC* model describes the correspondence to both the *join* and the *fork* nodes, and thus, it accounts for the syntax and semantics of both nodes. The *join* node can have multiple incoming flows and a single outgoing flow, while the *fork* node can have single incoming flows and multiple outgoing flows. The behavioral semantics of both nodes are captured in the specification of δ_{ext}, δ_{int}, δ_{con}, and λ functions. For structural semantics, the model has a list of queues, each corresponding to an input port for holding inputs while waiting for other inputs to arrive through other input ports. Once there is an element in each queue, the model moves into a transitory state to dispatch the output (Figure 3). We also define a mechanism to check the correspondence among these inputs.

The role of the ports in the *SYNC* atomic model captures the syntax of the *join* node. The in_i input ports and the out_i output port correspond to incoming and outgoing flows for the *join* node (see Figure 4 (c) and (d)). The behavior specification for the *SYNC* model (see Listing 1) shows the importance of providing structural and behavioral semantics for flows into and out of the UML activity node, and the inclusion of the ports for the *SYNC* model enhances coupling it to action nodes. The coupling with ports is more expressive as compared with flows that abstractly connect activity nodes. The same observation applies to the *SELECT* model.

2) SELECT ACTIVITY MODEL

Similar to the *SYNC* model, the *SELECT* model (see Listing 2) has one input port for each incoming flow and one output port for each outgoing flow. However, based on the semantics of the *decision* and *merge* activity nodes, the *SELECT* model transitions to an active state as soon as it gets input through one of its input ports. Then, based on specific or probabilistic conditions, it decides which port the output is dispatched from. After determination, the output

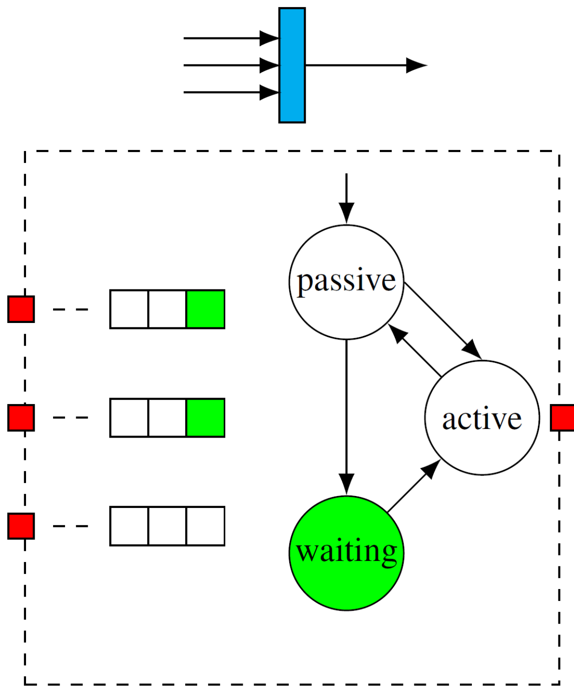


FIGURE 3. Illustrating the inner of the external transition function for a waiting state in the model of a *join* node shown at the top.

gets dispatched only through that particular output port. In the case of a merge, the output is always dispatched through the same output port since the model has only one. In other words, the *SELECT* model with multiple input ports and one output port corresponds to a *merge* node. Moreover, with a single input port and multiple output ports, it corresponds to a *decision* node. It also corresponds to both if it has multiple inputs and multiple output ports. Figure 4 illustrates basic mapping from activities to DEVS.

The *merge* and *decision* nodes are both control nodes, and they are symmetric in terms of their incoming and outgoing flows. The *merge* node receives multiple incoming flows and produces a single outgoing flow, while the *decision* node receives a single incoming flow and produces multiple outgoing flows. From a semantic point of view, they receive or produce flows that their guarding conditions evaluate as true. Only a single flow is selected for a particular I/O. Similar to the case in the *SYNC* model, the behavioral semantics of both the *merge* and *decision* nodes are captured in the *SELECT* model in the specifications of δ_{ext}, δ_{int}, δ_{con}, and λ functions. A list of Boolean values is attached to correspond to the flows for the evaluation. Also, a queue is defined for the case of holding elements when receiving inputs while in a busy state. Once an element is received, a transitory state is instantaneously entered before dispatching output.

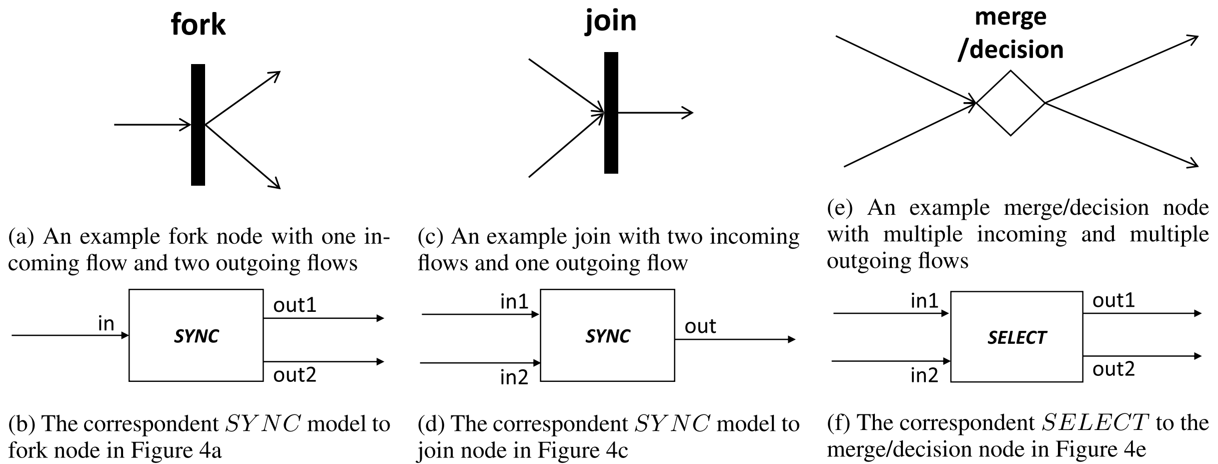
3) COUPLED ACTIVITY MODEL

In the DEVS formalism, each coupling is attached to the two ports defined at its ends. All couplings are uni-directional and there can be no self couplings. For internal coupling, the beginning of the coupling assigns to an output port, and the end assigns to an input port. It is, however, permissible for a port to be attached to multiple couplings. For example, a model A with one output port *out* can be attached to two couplings c_1 and c_2 where c_1 links the output port *out* in A to the input port *in* in model B, and c_2 links the output port *out* in A to input port *in* in model C. In such a mechanism, any output dispatched from model A will be duplicated and simultaneously sent out to both models B and C. This discipline may appear to be suitable for the *SYNC* specification and the *fork* node in particular. It allows for dividing or combining mechanisms in model A. Conversely, such a mechanism, once needed, ought to also be part of the receiving B and C models. Such a scenario is possible in this example, but it should not be imposed. Therefore, we propose dedicating a single output port for each coupling to allow for combining or dividing mechanisms to be defined in any one of models A, B, or C or any combinations thereof (see Figure 4). Hence, such mechanisms are left undefined in both DEVS and UML. They are both abstract in terms of requiring mechanisms to handle output getting dispatched through outgoing ports or pins (single or multiple) with multiple links attached to them (couplings or flows). In the parallel DEVS formalism, the receipt of multiple inputs through the same input port is possible. However, they operate in an arbitrary order if they arrive at the same time instant. In the UML, defining an input pin with multiple incoming flows imposes join-like semantics dictating that execution should wait for inputs from all incoming flows before proceeding [25].

In the following sections, we will examine the use of each of the *SYNC* and *SELECT* specifications in modeling the relevant activity control flows. In section V, we focus on the *SYNC* model. In section VI, we focus on the *SELECT* model. While in section VI-B, we discuss the use of both in the same flow.

V. EXPLOITING PARALLELISM

In the proposed activity specification, the parallel execution is employed based on the parallel, modular, and hierarchical DEVS formalism [27]. The implementation of the simulation algorithm determines whether to conduct the simulation procedure in a parallel or sequential manner while adhering to the strict principles offered in the formalism. The simulation is not necessarily executed in parallel, although we account for parallelism from the modeling standpoint, such as in the activity-based model. Parallel discrete event simulation (PDES) is divided into two categories [28]. First, conservative approaches are developed based on strictly preventing causality violations. The second category is optimistic, where the simulation allows violation of causality constraints while employing mechanisms to detect and resolve them when they happen. The Parallel DEVS simulator [2] exploits parallelism based on strict adherence to causality constraint. Events in some models can cause or otherwise influence events in other components. Due to modularity, the components only communicate events through input/output couplings. In each



(a) An example fork node with one incoming flow and two outgoing flows

(c) An example join with two incoming flows and one outgoing flow

(e) An example merge/decision node with multiple incoming and multiple outgoing flows

(b) The correspondent $SYNC$ model to fork node in Figure 4a

(d) The correspondent $SYNC$ model to join node in Figure 4c

(f) The correspondent $SELECT$ to the merge/decision node in Figure 4e

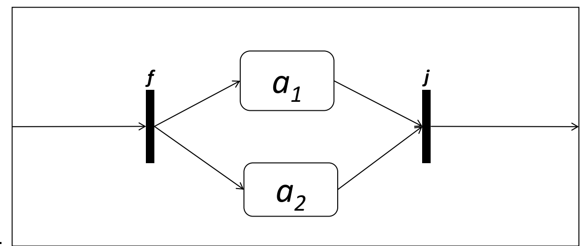
FIGURE 4. Illustration of the mapping of different activity nodes with accounts to multiple ports and couplings.

simulation cycle, all imminent components execute in parallel. Once all output events have taken place, all the corresponding influences will execute in parallel for the subsequent processing of input events (received output events). The simulation protocol maintains the variables for the time of the last event and the time of the next event along with output message bags to exchange among components through flat and hierarchical couplings.

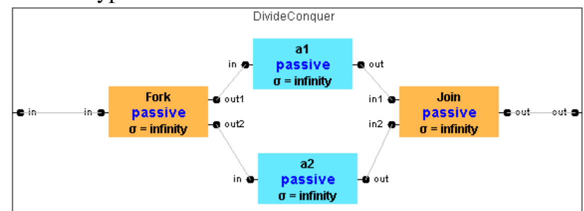
A. PARALLELISM SEMANTICS

The use of activities stands to serve as a basis for describing different patterns that include parallelism semantics. Encountering such situations is an essential part of developing activities, and therefore, should be examined in a simulation environment. In this section, an example (divide and conquer multiple processor archetype) is selected to demonstrate the use of the first type for control activity nodes (i.e., *fork* and *join* nodes), which demonstrate some aspects of parallelism semantics because they allow for parallel flows to proceed within an activity. They are both shaped with an opaque rectangle (see Figure 5). And to a limited degree, they also resemble the semantics of the *transition* concept in Petri nets [29]. The *fork* node is used to split an incoming flow into multiple concurrent outgoing flows. The offering of the flow can be arbitrarily accepted by the receiving nodes. The *join* node, on the other hand, receives multiple incoming flows and produces one after synchronizing them. The concurrent flows in a_1 and a_2 are two independent components. However, they synchronize at two junctions, at the flows of events into and out of the *fork* and *join* components.

Due to the modularity in the DEVS formalism, the representative components for semantics mapping are defined with input and output ports. An atomic model component generally has multiple input and output ports. The simultaneous arrival of a bag of inputs may occur through the same or different ports. In the case of having a coupling between one output port of a model and multiple input ports belonging to another model, the output is duplicated and sent to each of the



(a) An activity abstraction for the divide and conquer architecture archetype



(b) The initial simulation view of the divide and conquer architecture

FIGURE 5. Activity-based modeling of the divide and conquer architecture.

input ports. Although the DEVS formalism does not have a built-in mechanism to prevent the duplication of events, it can be accounted for in the model.

On the one hand, the *fork* node may dedicate an output port for each outgoing flow. On the other hand, one output port can correspond to all outgoing flows. In the former, distinct outputs are sent through different couplings. In the latter, an output is carried through all correspondent couplings, which can be replaced with some other logic in the model. The expressiveness, complexity, and scale of these approaches can be further examined.

Communicating I/O through coupling is instantaneous. Therefore, inputs arrive at the corresponding models in parallel (i.e., at the same time instance). Then, the model has the responsibility to process, hold, or lose any input it receives. Multiple inputs can be simultaneously obtained by the same

model. The processing can take place if the model is in a "passive" state (i.e., a state in which the model can accept inputs). The holding occurs if the model has some queuing mechanism. The inputs that cannot be processed or otherwise stored are lost.

Another essential aspect of parallelism is the handling of event collisions. The input may arrive at the same time instant when output is scheduled to dispatch. The confluence function δ_{con} can handle this collision between input and output. The order was imposed in the classic DEVS formalism using a *select* function where output dispatching precedes the processing of inputs. Moreover, only one atomic model of a coupled model must execute at once; however, in the parallel DEVS formalism [27], this restriction is relaxed. Every atomic model can specify the simultaneous input and output ordering and execution independently of any other atomic model because the constituent atomic models accommodate the simultaneity of the input and output events. They also account for the uni-directional external input and the internal and external output couplings.

B. A SIMPLE EXPERIMENT FOR AN ARCHETYPE DIVIDE AND CONQUER ARCHITECTURE IN THE DEVS-SUITE SIMULATOR

We devise an experiment to demonstrate some of the aspects discussed earlier with the DEVS-Suite simulator is used for developing and executing the experiment. The DEVS-Suite simulator is equipped with capabilities such as animations and linear and superdense time trajectory run-time tracking, and these capabilities are used to observe and monitor key aspects of the behavior of the archetype architecture. First, the divide and conquer architecture is coupled with an experimental frame (EF) model [30]. The EF has a simple generator to stimulate the archetype model by providing the inputs. For demonstration purposes, this simple experiment generates outputs every five-time units. The transducer is used to analyze the model's properties, such as turnaround time and throughput for processed jobs. The generator communicates with the divide and conquer coupled model (Figure 5b) via an external input coupling, and the transmission of a job through each coupling is instantaneous. It is easy to assign a delay to job transmission by, for example, introducing a delay component between the sender and receiver of the job. Alternatively, a delay can be added to the time assigned for processing the job. Once the job arrives at the fork node, the model sets its sigma to zero time advance. Therefore, it only transitions to a transitory state which instantaneously sends out the job. Essentially, a delay period can be set, as we will discuss further when making observations about timing. Nonetheless, we note that coupling in DEVS is instantaneous. The account for such delay can only be made through the notion of elapsed time or in the time advance function, which is defined within the atomic model.

A dedicated port corresponds to each coupling. However, the output is dispatched simultaneously by the output function in the *fork* component. The components a_1 and a_2 (i.e., processors) receive the inputs simultaneously. If a processor is in the phase *passive*, it transitions to the phase *busy*. To observe a certain behavior, a_1 is set to process inputs five times faster than a_2. Each processor has a FIFO queue to hold jobs, and the stored jobs are the ones received during phase *busy*. The *join* component is specified to receive both inputs after being processed by the a_1 and a_2 components. When input arrives on a port, it waits for an input from the other input port. The received inputs are combined, and then there is the possibility to add a delay before dispatching. Note that the state trajectory for the phase of a_2 (Figure 6) remains unchanged in phase *busy* because it processes jobs slower than a_1. The time needed for processing is greater than or equal to the job arrival rate. Therefore, a new job arrives before or immediately after finishing the current one. In this configuration, we set the processing time to be equal to the job arrival rate to illustrate superdense time trajectories, as shown in Figure 6 right beneath the phase time trajectory. In such a case, a phase repositioning to *passive* happens at the same time instant of receiving subsequent jobs and therefore, it transitions back to *busy*. Note that this repositioning does not appear in the main trajectory because it is instantaneous. The property of a_2 remaining in phase *busy* can be formulated and checked in tools with formal verification capabilities such as UPPAAL. The archetype can also be specified using constrained DEVS [31], and then verified using the DEVS-Suite. It indicates that once a_1 enters the phase *busy*, it remains in this phase forever, which is consistent with the state trajectory shown in Figure 6.

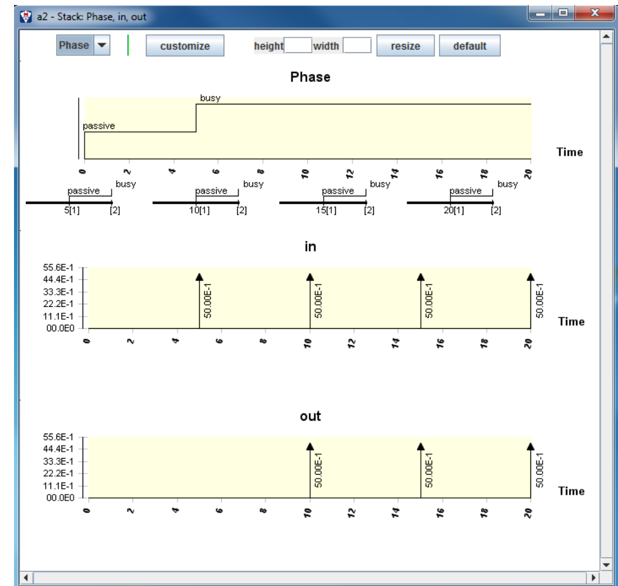
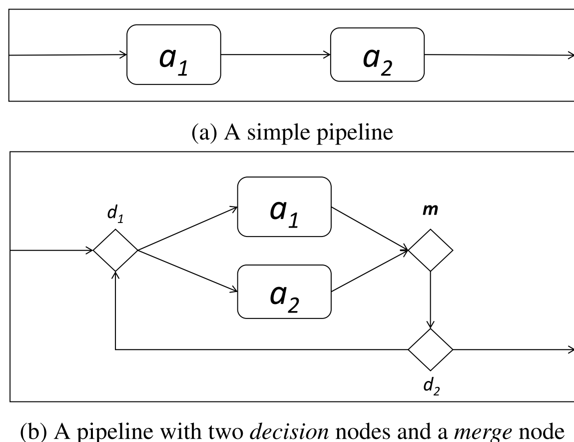


FIGURE 6. The trajectories for the state variable ***phase*** and the input ***in*** and output ***out*** ports with events for the a_2 component.

VI. FLOW SELECTION SCHEMES

In a multi-processor pipeline architecture, a job may travel through multiple stages before being completed. At each step, the model may have to decide whether the job has been



(a) A simple pipeline

(b) A pipeline with two *decision* nodes and a *merge* node

FIGURE 7. **Different abstractions of the pipeline architecture with possibly different temporal attribution in their simulations.**

completed or if additional processing is needed. A *decision* node in Figure 7 depicts this choice. Such a node only represents an abstract element for selecting one of its outgoing flows. It has yet to be equipped with certain conditions or perhaps a utility function to facilitate decision-making. In this architecture example, the *decision* node is concerned about the completion of the job.

Further considerations may take place in this decision logic. Activity diagrams alone do not stand to support such elaboration. The outgoing flows from the *decision* (choice) node are associated with abstract conditions that are left to be elaborated in one or more concrete layers following the MDA concept. In our approach, we examine concretization by developing a simulation of the *decision* node based on the DEVS formalism. Meanwhile, the separation between the abstract layer/layers and their counterpart concrete ones is iterative and thoroughly maintained.

We define the processing stage as a stage where a job undergoes partial processing accompanied by a delay. Since the job travels through multiple stages, its processing time is simply the total of the delays encountered at each processing stage. Hence, an activity is too abstract when it comes to the concept of time, and neither activity nodes nor edges can account for the delay in the UML metamodel 2.5 [16]. In DEVS formalism, the notion of the passage of time is supported for atomic models with dispatching and receiving of events between any two components occurring in order at the same time instant. We propose the use of control nodes to provide a means for the so-called controlled coupling [21], and we show that such a control node benefits from a more intuitive yet rigorous framework for modeling time-based dynamics of distributed systems.

A. A PIPELINE ARCHITECTURE

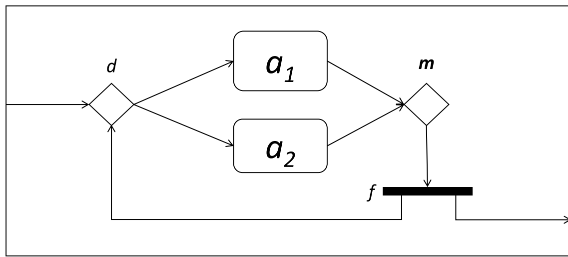
As described above, a simple pipeline consists of multiple units for processing a task (a non-trivial job) in a piecemeal fashion and in a particular order. Another way is to introduce decision points among units to determine when the task has been completed and to which unit the task needs to be assigned next. Many aspects of the feedforward and feedback disciplines (Figure 7) can be accounted for in both activity and DEVS models.

Considering activity modeling, a key to processing tasks as such is the way nodes are organized to allow the flow of a certain task. Such activity elements can exist in the flow. Each flow can also be characterized by the nodes that precede it and the nodes that follow it. For example, the outgoing flows from a *decision* node can relate to propositions that evaluate to either true or false. This is not necessarily the case with outgoing flows from an *action* or even a *fork* node. Multiple outgoing flows can be produced from the same node (e.g., a *fork* node) as described in the divide and conquer architecture. Hence, in the pipeline with feedback, parallel flows are allowed. In Figure 7a, we illustrate the discipline of a pipeline where the task travels through single flows to different elements in a strict sequential order. In Figure 7b, we illustrate the decision-making process, where each task also travels through the same elements while allowing parallel flows for multiple actions. In this architecture, a task encounters two decision-making procedures.

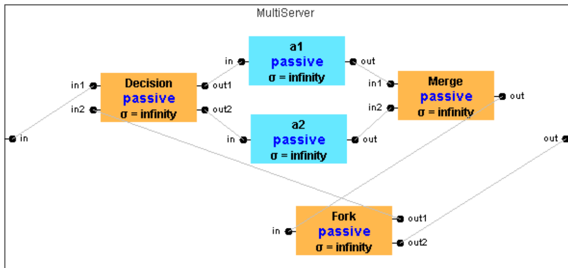
First, the task is checked before assignment to a certain processor. Second, whether or not the task has been completed is checked. A possible scenario of processing one task starts with the task being checked and assigned to a_1. After some delay, the task is sent out from a_1 to the *merge* node and immediately delivered to *decision* node d_2. Decision node d_2 checks whether the task has been completed or not. When the task has not been completed, it is sent back to d_1 and then assigned for processing at a_2. Afterward, this processed task directs to the *merge* node and immediately to the *decision* node d_2, which dispatches the completed task if it has completed processing the required action nodes. The two pipeline disciplines differ in their activity elements even though they deliver the same outcomes from the standpoint of timing and simulation. Having the discipline with two decision elements allows for further control over the assignment of tasks to different processors and checking for completion of the tasks. This discipline provides a greater degree of specification of time granularity through the *decision* and *merge* nodes (i.e., lifting the restriction on the choices for the incoming and outgoing flows to be instantaneous). Therefore, the *decision* and *merge* nodes can be used to devise different pipeline processing procedures. Furthermore, additional measures of performance, such as throughput and buffering, can be computed.

B. A MULTI-SERVER ARCHITECTURE

It follows from the previous two architectures, namely the divide and conquer and the pipeline that all of the essential activity nodes have been discussed and used in both multi-processing regimes. In this abstraction, the *decision*, *merge*, and *fork* nodes are used (see Figure 8). An activity input parameter defines the activity and is where the *decision* node d receives its incoming flows, and a decision is made



(a) An activity for a multi-server architecture



(b) The initial simulation view of the multi-server model

FIGURE 8. Activity-based modeling of the multi-server architecture.

to which subsequent node (i.e., a_1 or a_2) the task needs to be sent. Its outgoing flows are subject to satisfying some Boolean conditions. Satisfying the condition redirects the task to the most suitable nodes that can depend on action node availability (see section VII-C for simulating activities in DEVS-Suite).

Other considerations can be taken into that account to achieve different computational goals. After the completion of the set of actions, the *merge* node *m* collects the tasks, and an output is produced. It acts as a bridge point for the flow toward other nodes and separates the flows from the *fork* node *f* to avoid synchronization. Because the two processors are independent in this architecture, they do not have to be synchronized as opposed to other types. Therefore, the activity in Figure 8a demonstrates the way a flow can be allowed to proceed without waiting for another. The other alternative is to link outgoing flows from both a_1 and a_2 to f directly, but that would enforce waiting for both flows, whereas that is unnecessary for this particular architecture. The *fork* node receives the completed tasks and then produces two outgoing flows. One flow acts as a notification being directed back to the *decision* node to notify it of the task completion. Thus, the corresponding resource becomes available for processing another task. The other flow goes to the activity output parameter.

VII. A FRAMEWORK FOR ACTIVITY MODELING AND SIMULATION

In an earlier study [20], we proposed and developed mapping from the parallel DEVS formalism to the elements of the activity model. The core focus of the mapping was on representing the behavior of the atomic DEVS model. The illustrative metamodel in Figure 9 shows important aspects of this relationship, where both the DEVS and activity metamodels

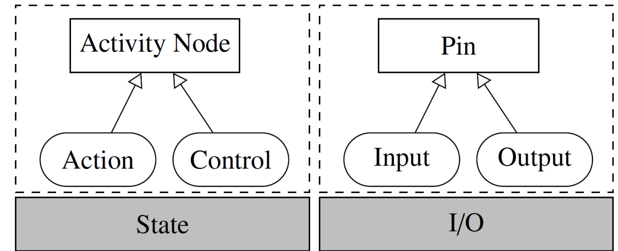


FIGURE 9. A high level sketch illustrating (i) the incorporation of the action and control nodes on the one hand and the state on the other and (ii) a conceptual relationship between the I/O and the activity pin.

are examined. On the one hand, the concept of state change as defined in DEVS (i.e., state transition, output, and time functions) is aligned to the activity node. On the other hand, various notions of the behavioral activity diagram as defined in the UML complement the atomic model. The I/O is also looked at from a DEVS vantage point to set the basis for the modular and hierarchical construction of models in the proposed approach.

A. TIME FOR ACTIVITIES

The simulation of activities such as those described above can be precise using a well-defined time base [19]. Explicit temporal specifications such as logical time can eliminate certain ambiguities that manifest themselves during execution. We also described some limitations related to the expressive behavior of the atomic DEVS model, on the one hand, and the precision of the activity model, on the other hand. It is difficult to account for limitations rooted in behavioral specification by solely depending on, for example, debugging code techniques and validation methods alone. Such defects particularly arise in behaviors with relatively more complex temporal structures such as the ones that must be characterized using superdense time [32].

When simulating the mentioned archetype above architectures, it is essential to observe certain phenomena, such as ordering and arrival of multiple inputs at any instance of time. Such aspects might result in state transitions that may not be readily traceable using typical software and simulation tools. Since architecture complexity is inherent, it is useful to have the means to generate superdense time trajectories for executable models [33] and for parallel DEVS models, in particular. This feature is necessary on multiple occasions, including the model development, testing, and simulation experiment for different multi-processor architectures. In all cases, the processing unit or any other component is expected to receive either single or multiple inputs simultaneously. It is also likely to have multiplicity and simultaneity for outputs.

When receiving multiple inputs, their order is arbitrary, and the model may or may not process the received inputs. The external transition function takes place given the current state, in addition to the received bag of inputs and the elapsed time. If a state transition is due to a single input, then the transition is visible using a linear time trajectory. The situation of zero

time advance, as in the divide and conquer archetype example in Figure 6, can be tracked using a superdense time trajectory. Such time representation is due to some elements that are added to control the flow without encumbering logical time. Therefore, state transitions that represent such elements can only be made visible using superdense time. The same holds true for similar nodes, especially the ones for control flow purposes only. We will discuss assigning logical execution time to different node types using the time advance function.

It is essential to consider various representations for the notion of time in a simulation environment, such as the ones presented by [34]. In some cases, limited time-based representations are inadequate when addressing relatively challenging concurrency and synchronization issues. We demonstrate the use of different taxonomies and how they may correspond to activities. The goal is to facilitate earlier experimentation for different processing architectures, with possibly varying lower-level manifestations through conforming to the MDA guidelines. Identifying a broader set of timing needs can be cumbersome. Thus, it is crucial to facilitate making a more informed decision, especially in cases where efficiency, cost, and scale trade-offs may exist.

B. OBSERVATIONS OF TEMPORAL ANALYSIS WITH ACTIVITIES

Notwithstanding the behavioral complexity detailed for multi-processor archetypes, a temporal analysis may follow, using an activity node classification based on the activity metamodel [16]. Different temporal aspects are characterized by which components are used to describe their specifications, and different temporal characteristics can be ascribed to components based on their specification aspects. For example, some components represent control nodes, such as decision. The time elapsed in these components is defined to be the time spent controlling the flow in some activity. Likewise, the time elapsed in other components is characterized based on the node type for which the specifications of these components ascribe.

Assume each node in the activity models mentioned above is associated with processing time pt, which is either zero or a positive real number. We refer to the activities in Figures 5a, 7b, and 8 as DC, PL, and MP, respectively. We also consider a task that is carried out by one activity holistically and can be assigned to one or divided among multiple activity processing nodes such as those defined for DC. The total time required for the task completion must be consumed in the processing nodes only. Assume the control nodes may consume time. However, this time cannot count toward task completion. Instead, they account for other time-consuming considerations such as overhead. We formulate such assumptions using the following definitions:

T_{pt} refers to the time required for completing the task or some part thereof.

T_p refers to the time from when processing the task/-tasks is initiated until its completion, without accounting for overhead or the time consumed for controlling the flows. Note that when the task is directed to one action only, then T_p is the same as T_{pt}. Also, note that when the task is assigned to N actions, then T_p will be equal to T_{pt}/N ad infinitum.

$c_{i,active}$ refers to the control node i while being active in managing the flow during the processing of the task.

$a_{i,active}$ refers to the action i while being active processing the task or a part thereof.

α is the task arrival rate.

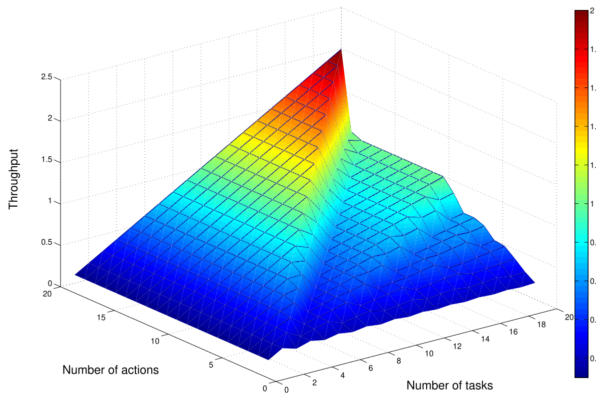
T_c refers to the time consumed by control nodes. Formally, for one component, it is the total time elapsed for the atomic model while in a phase with a specific finite time duration. Thus, T_c of all control nodes in an activity is the total finite time for all atomic models that correspond to these control nodes in the activity. This duration may account for dividing tasks into multiple sub-tasks and combining them if necessary. It may also account for synchronization and other timing considerations. Hence, time consumption varies from one architecture to another since the controlling mechanism may differ. In DC, T_c would include the time consumed by *fork* corresponding components and refers to the time required for dividing the task to prepare it for being processed by processor components. It also includes the time needed for combining multiple parts (the join) of the tasks after being processed. In PL, T_c would consist of the time consumed by the *decision* nodes. In the first *decision* node, it refers to the time required to decide which processing node the task needs to be directed. In the second one, it refers to the time needed for determining whether the task has been completed or not. The T_c also includes the required time for merging flows. In MP, T_c consists of the time required for deciding to which processing node the task needs to be assigned. It also includes the time needed for merging the flows and the time needed for redirecting the completed task and notifying the decision component of the task completion.

The task is processed in either a_1 or a_2, as in the MP architecture. The task could also be processed in parallel in a_1 and a_2, as in the DC architecture, or sequentially, as in the PL architecture. For the above archetypes, time consumption is calculated as $T_{pt}(task) = T_{pt}(a_{1,active}) + T_{pt}(a_{2,active})$. For an archetype with an arbitrary size, the time required for task completion is equal to the total active time of all actions A that carry out the processing, which can be formulated as

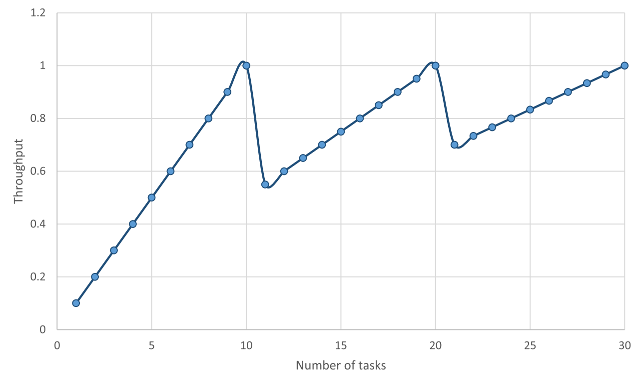
$$T_{pt}(task) = \sum_{i=1}^{n} T_{pt}(a_{i,active}), \quad (1)$$

where $n \in \mathbb{N}$ is the number of actions, for example two in the activities mentioned above.

Similarly, T_c is defined to allow accounting for the overhead time in different multi-processor architectures with different performance schemes. In Figure 5a, this accounts for the time consumed by the corresponding atomic models for the *fork* as well as the *join* nodes. For the given DC, the T_c for a given task is defined as $T_c(task) = T_c(c_{1,active}) + T_c(c_{2,active})$. Thus, the time consumed is equal to the total



(a) Throughput given different number of actions and tasks arriving at the same time.



(b) Throughput when the number of actions is set to ten ($n = 10$) given different number of tasks arriving at the same time.

FIGURE 10. Particular cases of throughput of the multi-processing (MP) and pipeline (PL) architecture are observed with different assignments of the number of tasks and actions. T_{pt} is set to 10 time units in all cases and T_c is assigned zero time unit.

active time of all control nodes, which can be formulated as

$$T_c(task) = \sum_{i=1}^{l} T_c(c_{i,active}), \tag{2}$$

where $l \in \mathbb{N}$ is the number of control nodes in that particular activity.

We assume the throughput can be measured based on both T_{pt} and T_c acquired from Eqs. 1 and 2, respectively. It can be used to identify the computational efficiency of each architecture while accounting for the distinction between processing time and other time-consuming elements. This type of difference is accessible through the abstraction of the meta-layer, where control nodes are being defined and then realized concretely in the simulation environment. Such measurements are essential for making critical decisions in various application domains. In formal terms, the throughput is identified based on the arrival and departure of the tasks to the coupled model that is created to correspond to the processing regime, with activity serving as an abstraction.

We characterize the time assigned by control nodes to be consumed by the task assigned to a particular node, dividing the task into sub-tasks or synchronizing sub-tasks. We assume all three are used in the DC architecture since the task has to be divided, assigned, and synchronized. In PL, only two of these mechanisms take place, the division, and the task assignment. In MP, only one takes place since the task has to get solely assigned to a particular node in a general multi-processing regime. Thus, a probability for T_c of each architecture is to be assigned relative to the number of actions.

Particular cases of throughput can be simply observed subject to restricted assumptions about the configuration of the experiment. For example, the number of tasks and their arrival rates is significant in determining the throughput, especially with the exploitation of parallelism. Under strict restrictions, some observations may follow trivially. For instance, the divide and conquer regime will outperform other architectures for one task if T_c consumes zero time in all architectures for each task. Similar observations can take place when different assumptions are given concerning other variables, such as the arrival rate of tasks or T_{pt}.

Other cases of throughput can be calculated under some strict assumptions for T_{pt} and the arrival rate of tasks. In the case of the DC architecture, the throughput can be calculated trivially based on these assumptions by simply dividing the number of actions by the total processing time for the completed tasks. In other words, the throughput is equal to n/T_{pt}. For the other archetypes (i.e., MP and PL), the case is less trivial due to the minimum sequential processing time. Nevertheless, throughput for these is the same as throughput for the DC architecture for the best-case scenario, where the number of tasks and the number of actions are equal (see Eq. 3). The worst-case scenario is where there is only one task that leads to less parallelism exploitation and, consequently, results in lower throughput. In Figure 10, some observations are made where T_p is calculated using the following formula:

$$T_p = \lceil \frac{k}{n} \rceil T_{pt}, \tag{3}$$

where k is the number of tasks, and n is the number of actions. Figure 10a shows throughput with different assignments for both n and k. Figure 10b shows throughput when $n = 10$ and with different assignments to k. Note that the use of the ceiling is due to the sequential part of the processing. This part has to be at least T_{pt}, resulting in throughput that is always at most one, and the cases of throughput get closer to one as the number of tasks increases, as shown. In the following section, we discuss other cases where the simulation becomes necessary to arrive at certain results.

C. SIMULATING ACTIVITIES IN THE DEVS-SUITE SIMULATOR

A set of atomic models corresponds to the discussed DEVS specifications of the activity constructs in the DEVS-Suite

simulator. The aim is to create models that complement the previously developed library and a tool for creating DEVS models with the activity notation. The library is made as generic and flexible as possible to allow it to account for a broader range of activity-based DEVS models. Currently, the library consists of two generic atomic models by which the primary activity control constructs can be realized in addition to the action.

The *decision* node is realized with multiple output ports and an array of Boolean values, where each value corresponds to a single output port. Upon the execution of the output function, some output is dispatched through a designated port when the Boolean condition that corresponds to the output is true. Hence, σ can become assigned with some positive value by δ_{ext} or δ_{int} functions in any atomic model that corresponds to the decision node of any other node. The *merge* node is similarly realized but with multiple input ports. Note that a node can be instantiated to have both decisions and *merge* properties as discussed earlier in Listing 2 for the *SELECT* model.

The *fork* and *join* nodes are also realized in the *SYNC* atomic model, where the implementation accounts for synchronizing input and outputs. Currently, the combining and dividing processes only account for the timing requirement. It remains an open problem to introduce a combining/dividing mechanism that may suit different semantics. The model corresponding to the *join* node includes multiple queues, where each queue accounts for a certain input port. Storing inputs in such a fashion permits accounting for inputs from different models that may arrive through multiple ports at different time instances. Listings 3 and 4 demonstrate the external and internal transition functions for the *fork/join* procedure, respectively.

```
parameter: double e, message x
begin
  continue(e)
  for i ← 1 to x.length
    job ← value of i_th message
    j ← port number
    add job to j_th queue
  end for
  if all queues are non−empty
      & phase is waiting then
    for i ← 1 to n
      dequeue from i_th queue
    end for
    holdIn(sending, prep_time)
  end if
  if phase is passive then
    holdIn(waiting, ∞)
  end if
end
```

LISTING 3. The external transition function of *SYNC*.

In the code snippets shown in the above listings, the atomic model for *join* is equipped with multiple queues to account for the multiple inputs arriving through different ports, along with their order. Note that this presumes the input flows conform to a particular order. When all queues are not empty, the first element of each is supposed to contribute to constituting the output that is to be dispatched. However, it is possible to prioritize elements of a specific queue based on heuristics. It is also possible to do further manipulation of the queue itself and its enqueue/dequeue procedures to satisfy different needs.

```
begin
  if all queues are empty then
    passivate
  else if all queues are non−empty then
    for i ← 1 to n
      dequeue from i_th queue
    end for
    holdIn(sending, prep_time)
  else
    holdIn(waiting, ∞)
  end if
end
```

LISTING 4. The internal transition function of *SYNC*.

We also devised an experiment for the DC architecture to observe throughput under different settings (i.e., the numbers for tasks and actions). In every setting, the experiment initiates by generating all the tasks instantaneously, with ten units of processing time for each task. It is possible to choose different configurations concerning arrival rates for tasks and processing time based on a specific distribution (e.g., uniform distribution). In this particular example, we set T_c relative to the number of actions, assuming more actions require more time to prepare for dividing and then combining sub-tasks. This setting amounts to the higher throughput encountered with more tasks and fewer actions (see Figure 11). The plot shows the best case, with ten tasks and two actions and worse throughput when there are more actions and fewer tasks. The throughput is calculated upon the finishing time, which is precisely the time unit of dispatching the last task by the corresponding coupled model of the DC activity. The final result gets calculated by merely dividing the number of tasks by that total time.

D. GENERATING DEVS MARKOV SIMULATION MODELS FOR PARALLEL ACTIVITY NODES

We can generate corresponding DEVS Markov models [35] for an activity with transition probabilities and time advances. Since we define states for actions, we can simulate any activity while giving a probabilistic time assignment for the next state based on Continuous-Time Markov (CTM) model. We devise an activity with ten parallel actions that correspond to processors and then run multiple simulations with varying probabilistic distribution and parallelism schemes. In the earlier demonstrations, deterministic timing assignments are assumed in the generated code for the models.

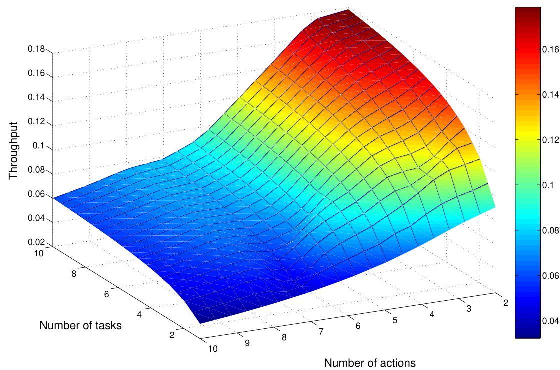


FIGURE 11. Throughput is observed by simulating the activity of divide and conquer in DEVS-Suite given different numbers of actions and tasks arriving at the same time. T_{pt} is equal to 10 time units in all cases and T_c is assigned linearly relative to the number of actions while a higher number of actions requires a higher T_c value.

TABLE 1. The result of simulating the generated Markov DEVS models corresponding to a fork-join scheme activities with ten parallel actions and 200 generated jobs.

Action	Number of lost jobs
1	37
2	30
3	41
4	42
5	33
6	46
7	30
8	29
9	37
10	39

In this experiment, we change the timing in each action as well as the generator using probabilistic modeling and distributions. For each action, we set the distribution for the time advance of the active state to be exponential and the mean to 10. We also set the distribution for generating the state of the generator to be uniform, with the lower and upper values set to 10. Table 1 shows the results of simulating the generated Markov DEVS models corresponding to a fork-join scheme activity with ten parallel actions. Since the actions in this model do not have queues or the means to store the arriving jobs when busy, the table shows the lost jobs by each action after simulating the model with 200 generated jobs. Note that only 23% of the jobs are lost when assuming the job are lost when arriving in a busy state that is implementing actions without a storing mechanism jobs to be processed later. Modelers can change the selections in this model. For example, it is possible to change the distribution setting or the assigned values for the transition. This example has been simulated using the MS4 Me simulator [36] because code generation for DEVS Markov models is supported for this tool. We plan to support generating more probabilistic models to be simulated in both the DEVS-Suite and MS4 Me simulators.

VIII. CONCLUSION

In this paper, we discussed within a meta-layer, a variety of elements can be proposed and connected in many ways. However, the inclusion of such elements might be ineffective or ambiguous when it comes to concrete realizations or used for transformation to concrete elements. Conversely, lower-level implementations cannot benefit from rich simulations without extensive modeling effort from a small set of higher-level elements. Often many iterations are necessary to find abstractions that are useful for creating concrete simulations. We devised a subset of the activity metamodel from the perspective of discrete event systems and discrete-event modeling. We started with a taxonomy of simulation modeling based on MDA and characterized different constructs of activity modeling based on their actual behaviors. We then devised a formal specification for each fundamental element and its corresponding implementation in the DEVS-Suite simulator. Different time notions are used to facilitate various temporal analyses. We demonstrated the distinction between control time and processing time. The characterization of activity constructs was used to classify and distinguish between their timing requirements and constraints. We showed the use of activity modeling for different multi-processing architectures. Our future work is on extending the framework and tools to support model verification and simulation validation, testing, experimentation, and activity-based modeling for hybrid systems.

REFERENCES

[1] A. W. Wymore, *Model-Based Systems Engineering*, vol. 3. Boca Raton, FL, USA: CRC Press, 1993.
[2] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. New York, NY, USA: Academic, 2018.
[3] R. Alur, *Principles of Cyber-Physical Systems*. Cambridge, MA, USA: MIT Press, 2015.
[4] H. S. Sarjoughian, A. Alshareef, and Y. Lei, "Behavioral DEVS metamodeling," in *Proc. Winter Simulation Conf. (WSC)*, Piscataway, NJ, USA, Dec. 2015, pp. 2788–2799.
[5] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. London, U.K.: Pearson, 2008.
[6] F. Fondement, P.-A. Muller, L. Thiry, B. Wittmann, and G. Forestier, "Big metamodels are evil," in *Model-Driven Engineering Languages and Systems*. Berlin, Germany: Springer, 2013, pp. 138–153.
[7] B. Zeigler, S. Mittal, and M. Traore, "MBSE with/out simulation: State of the art and way forward," *Systems*, vol. 6, no. 4, p. 40, Nov. 2018.
[8] D. Harel and M. Politi, *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*. New York, NY, USA: McGraw-Hill, 1998.
[9] H. Störrle, J. H. Hausmann, and U. Paderborn, "Towards a formal semantics of UML 2.0 activities," in *Proc. German Softw. Eng. Conf.*, 2005, pp. 117–128.
[10] L. Yonglin, W. Weiping, L. Qun, and Z. Yifan, "A transformation model from DEVS to SMP2 based on MDA," *Simulation Modeling Pract. Theory*, vol. 17, no. 10, pp. 1690–1709, Nov. 2009.
[11] J. L. Risco-Martín, J. M. de la Cruz, S. Mittal, and B. P. Zeigler, "EUDEVS: Executable UML with DEVS theory of modeling and simulation," *Simulation*, vol. 85, nos. 11–12, pp. 750–777, Nov. 2009.
[12] G.-D. Kapos, V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos, "An integrated framework for automated simulation of SysML models using DEVS," *Simulation*, vol. 90, no. 6, pp. 717–744, Jun. 2014.
[13] P. Bocciarelli, A. D'Ambrogio, A. Falcone, A. Garro, and A. Giglio, "A model-driven approach to enable the simulation of complex systems on distributed architectures," *Simulation*, vol. 95, no. 12, pp. 1185–1211, Dec. 2019.
[14] D. Foures, V. Albert, J. C. Pascal, and A. Nketsa, "Automation of SysML activity diagram simulation with model-driven engineering approach," in *Proc. Symp. Modeling Simulation-DEVS Integrative M S Symp.*, 2012, pp. 1–6.
[15] Richard Soley and the OMG Staff Strategy Group, "Model driven architecture," OMG, Needham, MA, USA, White Paper 308, 2000.

[16] *Unified Modeling Language Version 2.5.1*, OMG, Needham, MA, USA, 2017.

[17] M. Nikolaidou, G.-D. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos, "Challenges in SysML model simulation," *Adv. Comput. Sci., Int. J.*, vol. 5, no. 4, pp. 49–56, 2016.

[18] B. P. Zeigler, J. W. Marvin, and J. J. Cadigan, "Systems engineering and simulation: Converging toward noble causes," in *Proc. Winter Simulation Conf. (WSC)*, Dec. 2018, pp. 3742–3752.

[19] A. Alshareef and S. Hessam Sarjoughian, "Parallelism semantics in modeling activities," in *Proc. Model. Simulation Symp., SpringSim (TMS)*, Baltimore, MD, USA, Apr. 2018, pp. 6:1–6:12.

[20] A. Alshareef and S. H. Sarjoughian, "DEVS specification for modeling and simulation of the UML activities," in *Proc. Symp. Model-Driven Approaches Simulation Eng. (Mod Sim)*, Virginia Beach, VA, USA, 2017, pp. 9:1–9:12.

[21] A. Alshareef, H. S. Sarjoughian, and B. Zarrin, "Activity-based DEVS modeling," *Simul. Model. Pract. Theory*, vol. 82, pp. 116–131, Mar. 2018.

[22] T. Mayerhofer, "Testing and debugging UML models based on fUML," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 1579–1582.

[23] A. Alshareef and H. Sarjoughian, "Metamodeling activities for hierarchical component-based models," in *Proc. Spring Simulation Conf. (SpringSim)*, Tucson, AZ, USA, Apr. 2019, pp. 1–12.

[24] *Semantics of a Foundational Subset for Executable UML Models (fUML) Version 1.4*, OMG, 2018.

[25] Eclipse Foundation. (2016). *Papyrus Mars release (1.1.3)*. [Online]. Available: https://eclipse.org/papyrus/

[26] ACIMS. (2021). *DEVS-Suite*. [Online]. Available: https://acims.asu.edu/software/devs-suite/version6.1.0

[27] A. C. H. Chow, "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *Trans. Soc. Comput. Simul.*, vol. 13, no. 2, pp. 55–68, 1996.

[28] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, vol. 300. New York, NY, USA: Wiley, 2000.

[29] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[30] J. W. Rozenblit, "Experimental frame specification methodology for hierarchical simulation modeling," *Int. J. Gen. Syst.*, vol. 19, no. 3, pp. 317–336, Oct. 1991.

[31] S. Gholami and S. H. Sarjoughian, "Modeling and verification of network-on-chip using constrained-DEVS," in *Proc. Symp. Theory Modeling Simulation*, Virginia Beach, VA, USA, Apr. 2017, pp. 9:1–9:12.

[32] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York, NY, USA: Springer, 1992.

[33] H. S. Sarjoughian and S. Sundaramoorthi, "Superdense time trajectories for DEVS simulation models," in *Proc. Symp. Theory Modeling Simulation: DEVS Integrative M&S Symp.*, 2015, pp. 249–256.

[34] R. Goldstein and A. Khan, "A taxonomy of event time representations," in *Proc. Symp. Theory Model. Simulation*, 2017, p. 6.

[35] C. Seo, B. P. Zeigler, and D. Kim, "DEVS Markov modeling and simulation: Formal definition and implementation," in *Proc. 4th ACM Interfaces Conf. Eng. Sci.*, 2018, pp. 1–12.

[36] MS4 Systems. (Accessed: Apr. 1, 2020). *MS4 Me Simulator Version 3.0, 2018*. [Online]. Available: http://ms4systems.com/pages/ms4me.php

ABDURRAHMAN ALSHAREEF was born in Riyadh, Saudi Arabia. He received the Ph.D. degree in computer science from Arizona State University, Tempe, in 2019.

He is currently an Assistant Professor at the College of Computer and Information Sciences, King Saud University, Riyadh. His research interests include discrete event system model development and simulation, activity-based modeling, model-driven engineering, software architecture, and metamodeling.

Dr. Alshareef is a Lifetime Member of the Society for Modeling and Simulation International (SCS) and a member of the Arizona Center for Integrative Modeling and Simulation (ACIMS). He was a recipient of the King Saud University Scholarship for master's and Ph.D. degrees. He was also a recipient of travel grants from the Saudi Arabia Cultural Mission, USA; the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University; and the Society for Modeling and Simulation International to show his research in different conferences and venues.

HESSAM S. SARJOUGHIAN received the B.S. degree from Mississippi State University, in 1984, and the M.S. and Ph.D. degrees from the University of Arizona, in 1988 and 1995, respectively, all in electrical and computer engineering.

He is an Associate Professor and the Co-Director of the Arizona Center for Integrative Modeling and Simulation, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA. He has been leading the DEVS-Suite simulator being used at universities and research institutes in numerous countries across all continents. He is the Co-Founder of the Arizona Center for Integrative Modeling and Simulation, a Founding Member of the Certified Modeling and Simulation Profession, and a Certified Modeling and Simulation Professional. His research interests include agent-based modeling, model theory, multi-formalism modeling, simulation-based design, software architecture and their use for modeling, simulating, and evaluating hybrid computational-physical systems of systems.

Dr. Sarjoughian has received four best and two best runner-up technical paper awards, one best poster award, the SCS Distinguished Service Award, the DEVS M&S First Place Award. He was a recipient of the Honorable Mention from the Defense Modeling and Simulation Office, currently named the Modeling and Simulation Coordination Office. He is an Associate Editor of the *SIMULATION: Transactions of the Society for Modeling and Simulation* journal.