Simulation Modelling Practice and Theory 82 (2018) 116–131

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Activity-based DEVS modeling

Abdurrahman Alshareef[a], Hessam S. Sarjoughian[a,*], Bahram Zarrin[a,b]

[a] Arizona Center for Integrative Modeling & Simulation, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, 699 S. Mill Avenue, Tempe, AZ, 85281, United States
[b] DTU Compute, Technical University of Denmark, Kgs Lyngby 2800, Denmark

ARTICLE INFO

Article history:
Received 13 May 2017
Revised 16 November 2017
Accepted 12 December 2017
Available online 29 December 2017

Keywords:
Activity modeling
Behavioral modeling
GMF
Model Driven Development
DEVS
UML

ABSTRACT

Use of model-driven approaches has been increasing to significantly benefit the process of building complex systems. Recently, an approach for specifying model behavior using UML activities has been devised to support the creation of DEVS models in a disciplined manner based on the model driven architecture and the UML concepts. In this paper, we further this work by grounding Activity-based DEVS modeling and developing a fully-fledged modeling engine to demonstrate applicability. We also detail the relevant aspects of the created metamodel in terms of modeling and simulation. A significant number of the artifacts of the UML 2.5 activities and actions, from the vantage point of DEVS behavioral modeling, is covered in details. Their semantics are discussed to the extent of time-accurate requirements for simulation. We characterize them in correspondence with the specification of the atomic model behavior. We demonstrate the approach with simple, yet expressive DEVS models.

1. Introduction

Constructing simulation models, despite from being quite costly and complex, remains indispensable and highly beneficial, especially for systems that do not lend themselves to analytical methods. System dynamics need to be determined in enough details to sufficiently address its different aspects under study in order to ultimately attain the potential benefit. The models have to be then realized in certain computational and physical environments in order to enable the simulation and experimentation thereafter. As system complexity grows, so does the importance of behavioral modeling. There are existing concepts and techniques where the structure modeling can be handled systematically to account for further system complexity and growth. However, these techniques fall short with respect to behavioral modeling. Increasingly behavioral models are becoming large and therefore difficult to understand, formulate, and maintain using conceptual (informal) and mathematical modeling as well as their implementation in programming languages and evaluations.

The Discrete EVent System specification (DEVS) formalism [1] can effectively serve as a basis for simulation-based design and formulation of modular, component-based system models. The parallel DEVS formalism, based on time, input, output, states, and state transition, is widely supported by simulators, DEVS-Suite [2] for example, that have been implemented in different computing environments. Simulators need to serve different needs through complementary advanced capabilities including action-based behavior specification. The input for simulators is mainly models although the simulators significantly differ in the form by which the models have to be formulated, simulated, and evaluated. This has led to the rise of utilizing

* Corresponding author.
E-mail addresses: alshareef@asu.edu (A. Alshareef), sarjoughian@asu.edu, hessam.sarjoughian@asu.edu (H.S. Sarjoughian), baza@dtu.dk (B. Zarrin).

https://doi.org/10.1016/j.simpat.2017.12.009
1569-190X/

the so-called Model-Driven Engineering in simulation especially with respect to the focus on creating platform-independent models. Models of this nature are less inclined to carry details specific to the execution environments and therefore can be used and maintained for a wider set of M&S platforms, a key benefit of Model-Driven Architecture (MDA) [3]. The definitions in this approach have been proven to be consistent with the theory of modeling and simulation in multiple occasions [4–8]. In fact, it is desirable to have models of this nature in order to allow modelers to be more focused on the problem and solution specifications in more neutral terms with respect to specific details of simulation frameworks. Behavior specification is not as simple as structure specification especially when system dynamics require understanding and formulation beyond conditional state changes and event handling. This fact is accounted for the recent approaches that adopt some of the other languages and formalisms (e.g., UML state machines) with capabilities that can afford specifying complex behaviors.

While some behavioral languages are adopted for the specification of DEVS atomic models, they significantly differ in their suitability, complexity, and provided capabilities. In this work, we attempt to dig deep into the activity modeling as a major modeling approach for the DEVS atomic model and by extension coupled model behavioral specifications. The approach is gaining more attention given its promising prospects for enhancing system modeling in multiple domains. Activity metamodel has also undergone major advances in the recent decade especially the release of the UML 2.0 [9] and the foundational subset of UML (fUML) [10]. The idea essentially is to adopt the UML activities for the behavioral specification of the DEVS atomic model according to the state of the art standards. Activities provide some unique capabilities with respect to other behavioral diagrams. We want to leverage them in a way that shortens the distance between the concrete models and their mathematical abstractions. The handling of actions in the activities gives a premise to overcome some of the behavioral modeling difficulties in general and the ones encountered in the other behavioral approaches (e.g., finite-state machines). A richer specification can be achieved when modelers consider a variety of behavioral specifications that better serve their needs.

We will discuss some necessary background in this subject. We will also compare and contrast our contribution with some of the existing approaches toward meeting the need for expressive behavioral modeling. We elaborate on the selected approach based on previous work [11] in conjunction with further discussion and alignment with the DEVS formalism. A modeling engine is created to manifest that at the implementation level alongside with processor with queue model as an exemplar.

2. Background

There exist formalisms, modeling languages, and frameworks to develop behavioral models. Our work is centered on the rigorous specification of DEVS as an abstract mathematical formalism accompanied with a framework supported with modeling languages and run-time execution. In the following sub-sections, we describe basic background details for understanding and developing behavioral models.

2.1. Parallel DEVS atomic model

The set-theoretic specification of the atomic model is an abstract representation of a standalone component of a system [1]. The formal specification can be defined independent of any language, and more generally simulation platforms. From a software standpoint, we need to have the specifications to be formulated in terms of a modeling and software programming languages. There are many DEVS simulators that can accept the specification of a model following a target simulator's programming language syntax, semantics, and specialized constructs such as model initialization. Examples of these tools are DEVS-Suite [2] and CoSMoS (Component-based System Modeling and Simulation) [12] where the programming language is Java. Other simulators use different languages as an input such as CD++ [13] and PowerDEVS [14] where the programming language is C++. In [15], the work provides a specific language based on the formal specification definition language with set of rules to translate it into simulatable models targeting simulators like DEVS-Suite and PowerDEVS. As defined in [1], the basic formalism of parallel DEVS model is an algebraic structure – atomic model = $\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$. X is the set of input events. S is state representing the tuple of sequential states. The state must have at least two independent variables. One is called *sigma* (σ), the time duration allocated to the current state of the model. The other variable, called *phase*, represents a set of state values that change and be tracked. Y is the set of output events. δ_{int} and δ_{ext} are the internal and external transition functions, respectively. The model receives a bag of inputs meaning that the elements of the bag may have multiple occurrences and have no ordering. The receiving model accounts for this possibility in order to perform a proper handling of the inputs. δ_{con} is the confluent transition function which can be specified to handle the collision between external and internal events. λ is the output function which transforms S into Y at arbitrary time instances. ta is the time advance function which maps the internal state into a positive real number using elapsed time since last state transition (i.e., it computes σ which can range from zero to infinity, inclusive). Any domain specific definition of the aforementioned functions must satisfy their corresponding abstract definitions as provided in the modeling formalism. Together the elements of the DEVS specification allow modeler to flexibly define operations and controls for system structure and behavior.

2.1.1. Simple processor

This example is selected to demonstrate some concepts throughout the discussion. We start with a simple processor and later extend to have a queue to demonstrate behavioral expressiveness. The simple processor only stores jobs upon their arrival, process them for some amount of time (duration), and then sends them through output port. It does not account for input buffering and pre-emption of a job that is being processed. The process behavioral specification as described in the DEVS formalism are devised in a set of activity models as an intermediary phase between them and their concrete manifestations. The simple processor example as presented by Zeigler and Sarjoughian [16] is defined as

$Processor_{processing\_time} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$

where

$IPorts = \{in\}, where \quad X_{in} = J$ (a set of job identifiers)

$X_M = \{(p, v) | p \in IPorts, v \in X_{in}\}$ is the set of input ports and values

$S = \{passive, busy\} \times \mathbb{R}_0^{+\infty} \times J$

$OPorts = \{out\}, where \quad Y_{out} = J$

$Y_M = \{(p, v) | p \in OPorts, v \in Y_{out}\}$ is the set of output ports and values

$\delta_{int}(phase, \sigma, x) = (passive, \infty, x)$

$\delta_{ext}((phase, \sigma, x), e, ((in, j_1), (in, j_2), \ldots, (in, j_n))), \quad j_i \in J_{in}$

$= ((busy, processing\_time), j_1, j_2, \ldots, j_n) \quad \text{if phase} = passive$

$= ((phase, \sigma - e), x) \quad \text{otherwise}$

$\delta_{con}((s, ta(s)), x) = \delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(busy, \sigma, j) = j$

$ta(phase, \sigma, j) = \sigma.$

2.2. Statecharts

There are many modeling languages available for specifying the behaviors of atomic DEVS models. Statecharts is popular due to its expressiveness power for representing complex behaviors of systems [17]. Statecharts defines mainly hierarchical states and state transitions. They can be used to specify discrete behavior of a system and its components. Other languages and metamodels also exist for modeling the behavior such as behavior diagrams in UML [18]. They provide unique notations where behavior can be captured in different diagrams. Each diagram generally has some advantages relative to some other diagrams. The diagrams vary in their syntax as well as semantics in order to satisfy different needs. One major diagram is UML state machines, which is considered to be a variant of Harel's statecharts. The transitions, as well as states, can be associated with some behavior. Modelers can use state machines, interactions, sequences, or activities for describing behaviors within and across model components.

2.3. Activities and actions

The UML Activities diagram is major method for developing detailed behavioral models [18]. Their standardized specifications including visual syntax and semantics have undergone significant changes under the stewardship of the International OMG Standardization consortium. Activities allow for behaviors to be specified using a set of elements along with their sequencing defined as control and object flows. The elements are defined as activity nodes. A node can be control, object, or executable node. Control nodes define the flow of tokens between activity nodes. A control node can be either *initial* to define the starting point of an activity execution, *final* to define when activity stops, *fork* for splitting a flow into multiple flows, *join* for synchronizing multiple flows, *merge* to act similar to join but without synchronizing flows, or *decision* to select between its outgoing flows. Each control node can be used to define certain behavioral properties of components such as the DEVS atomic model. Object nodes are used to handle data. We will use activity parameter node to define inputs. Finally, executable nodes are the core elements of activities. Actions are defined as executable nodes and therefore they can be created within an activity. On the other hand, actions in UML 2.5 [18] can be only defined in the context of an activity. Thus, together they provide a means for modeling behavior to define processing routines that include control structures. It is important to note that activity modeling emphasizes and support specifying actions and combining them using arbitrary control structures.

Actions are the only kind of the executable nodes in UML 2.5. They are necessary to be used in order to take advantage of more capabilities provided by the activities. In addition, they are the fundamental units of behavior specification. Some actions change the state of the system. This kind of actions in our approach satisfies how states are changed in DEVS atomic model internal and external transition functions. Examples of these actions are *add structural feature value* and *value specification* actions. Other kinds of action support handling objects. *Read self* action is used to obtain the current object context and place it in its output pin. *Value specification* action is also used to provide certain value and place it in its output pin. The structural feature actions are used for either assigning or retrieving a structural feature of an object. They can be both used for the *phase* as a structural feature. They can be also used for other features or more complex objects.

Event actions can be also used. For example, an *accept event* action can be used to model the waiting for an event to occur in some other entity to proceed in the activity flow. In the context of UML actions, this event can be caused by the simulation protocol to trigger some atomic model components, or by other models that decompose the behavior into

multiple models. There can be also an *accept time event* action which is basically used to model waiting time. *Send signal* action can be used to model invocation for some other components. It is also used to explicitly enforce some order when used with the accept event. For example, to enforce the order of executing the DEVS output and internal transition functions, a sending event signal must be completed to enable the accept event action although this sort of scenarios could be viewed as part of the simulation protocol.

Additionally, invocation actions provide a means for communication and signaling among multiple activities, e.g. *call behavior* action, to call either a behavior or an operation. The action can be synchronous if it has to wait for the called behavior or operation to be completed. Otherwise, it can be asynchronous and then immediately proceed after calling the associated behavior or operation. This is crucial for the execution of the behavior of the atomic model (i.e., internal and external transition functions cannot be executed simultaneously although possible using the confluent transition function). We will elaborate further in the subsequent sections.

2.4. Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [19] is the core of Eclipse modeling project which serves as a basis for modeling and metamodeling. The framework and its various capabilities provide an Eclipse realized manifestation of the Model-Driven Architecture (MDA). It is surrounded and equipped with a variety of tools to facilitate the process of creating and maintaining metamodels as well as producing sets of classes and runtime support in a highly model-driven development. The Ecore as the basis for EMF is used to define metamodels to provide a common grounding for UML, XMI, and Java. It unifies these and other models in a well-defined setting with automated mapping from one to another (e.g., UML to Java). This is important as it places strong focus on the model as the fundamental unit for building software-based systems.

EMF has been extended to provide a means for creating metamodels for parallel DEVS formalism as in EMF-DEVS [6]. The framework currently supports defining and validating the structure of atomic and coupled meta-DEVS models. As discussed earlier, we want to take a step forward into behavioral modeling which turned out to be quite complex since EMF itself is mainly concerned with structural aspects. However, in a previous work [8], we discuss behavioral metamodeling for DEVS by extending the Ecore. In this work, we realize the activity metamodel as a major step in creating a behavioral modeling engine and preparing for further support such as defining the graphical definition and mapping.

Graphical Modeling Framework (GMF) [20] has been built upon EMF. It exploits EMF capabilities such as code generation and model serialization to visually specify models that conform to their metamodels defined in Ecore. GMF, as an extension of EMF, allows for building tooling infrastructure to be used for modeling and diagram generation. This approach is used in developing an engine enabling specification of statecharts for DEVS atomic models [21]. Although this approach uses both EMF and the Graphical Editing Framework (GEF), it segregates the metamodel from the graphical information in a well-defined manner. This is especially useful in our case since we need to support graphical notations for activities as well as maintaining consistency between the structural and behavioral metamodels for DEVS modeling. The metamodel is referred to as a domain model in the context of GMF. Once defined, there are two models to be initially generated and then manipulated to account for the specific graphical requirements. Those two models are the graphical definition model to define graphical components and figures used in the models and the tooling definition model which is used basically to define the other aspects of the editor such as the palette. The generation model is generated similarly with EMF. After that, the mapping model combines all definitions in the three previous models to put things together and map every element to its graphical counterparts. Once created, the process of generating the diagram editor becomes ready to be performed.

3. Related work

The notion of behavioral modeling has been employed by researchers for different purposes. As far as we know, it has been employed in order to bridge some of the distance between the mathematical specification of system-theoretic model in the DEVS formalism and their counterpart incarnations in computational forms. We characterize these efforts in two major categories. The first one is a state-based approach such as statecharts. The second one is a flow-based approach such as activities. Both approaches are considered to be major behavioral diagrams in the realm of the UML 2.5 which we attempt to align with as much as possible without compromising rigor. The state-based approach has been employed by many [4,22–24] in order to support defining behavior in the form of statecharts or state machines. These add new means to define behavior for DEVS atomic models. Furthermore, this approach provides graphical model development and mechanisms for validating models without execution, for example, using EMF [6,21]. Another related work is the Action-Level Real-Time (ALRT) DEVS modeling where the external and internal transition functions are formalized in terms of not only state, but also actions having their own timings and executed using a real-time simulation protocol [25]. However, this variant of the parallel DEVS formalism is not based on UML metamodel concept and the activity modeling.

Since some existing methods (statecharts and state machines) are aimed at certain aspects of behavioral modeling, the use of other methods becomes quite compelling. This explains the tendency to use activity modeling as an approach to aid development of DEVS models. The use of activities for the DEVS atomic functions (δ_{ext}, δ_{int}, δ_{con}, λ, ta) complements other behavioral specifications with certain capabilities such as ordering and the containment of actions within a UML context. In addition, there has been recent advances especially with respect to model execution using the foundational UML subset [10]. Activity modeling is also used for SysML models [26,27] as well as developing model abstractions for application-specific

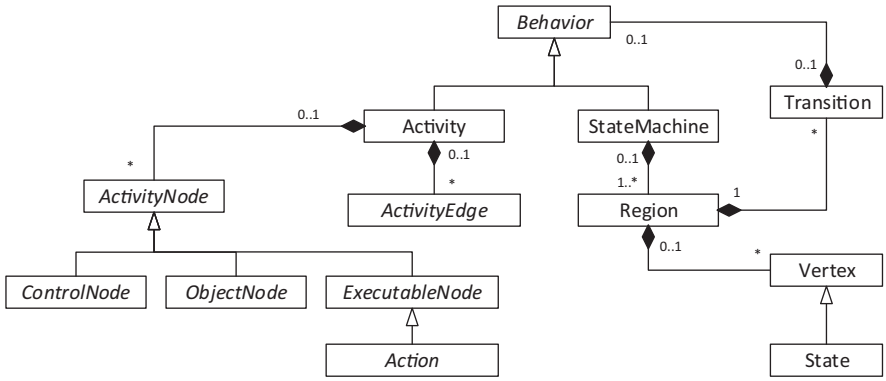


Fig. 1. A subset of behavior elements and their relationships in UML 2.5 metamodel.

domains [28]. In the latter, activity modeling is purposed for simulation protocol, but not, for example, in terms of actions and controls within atomic models.

In this work, we consider activities as a complementary modeling approach for visual behavioral specification of the parallel DEVS atomic model. We specialize the activity metamodel to serve as a basis for action-level behavior modeling as well as proposing and developing a modeling engine for parallel DEVS atomic modeling (see Section 5). The activity models are graphically developed utilizing the capabilities provided with Eclipse Modeling Framework (EMF) and its Graphical Modeling Framework (GMF). We create an Ecore metamodel to encompass the defined elements and constructs according to the activity modeling approach. Our objective is to account for as much details as possible in terms of the behavioral specification relative to level of details used in simulators. At the same time, we account for the principles defined in the Model Driven Development and Model Driven Architecture in order to enable the approach to be expanded further to incur the expected complexity that may arise during the model development with respect to behavior. The ability to come across some level of control over the different level of abstractions in such a grounding can be highly beneficial.

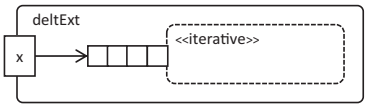
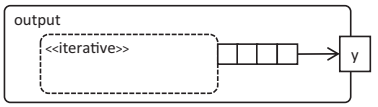
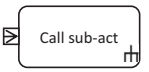
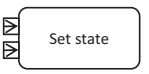
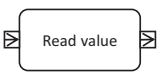
4. Activity-based DEVS modeling approach

We thoroughly investigate the use of activities in DEVS modeling starting by bringing into a common perspective the DEVS formalism, behavioral specification, and UML behavioral diagrams. We show where activities can come into place within existing DEVS metamodels (i.e., DEVS to SMP2 [29], MDD4MS [5], and EMF-DEVS [6]). We also show where they can come into place by benefiting from model-driven approach and considering the other behavioral diagrams, specifically state machines. Then, we discuss possible views to employ activities for DEVS modeling and then elaborate on these views to establish the selected approach.

The functions δ_{ext}, δ_{int}, δ_{con}, λ, and *ta* are defined in a DEVS metamodel [6]. These functions capture both the abstract structure and behavior of the formal DEVS atomic model specification. Each function has one or more elements defining its structure. For example, the structure of external transition function is defined strictly to only have input events and state while the structure of the output function is defined to have only output events and state. The behavior of these functions define what in the model can be changed (allowed behaviors). For example, in the external transition function, the state of the model can change, but in the output function state change is not allowed. Therefore, the behavioral specifications can be defined at the meta-layer to represent a system's behavior, for example, state machines. The activities can be also used to define the syntax for aforementioned functions. In the UML state machines, a variant of the original statecharts formalism [17], the metamodel of the UML 2.5 associates the *behavior* element with the *state* and *transition* elements. *Behavior* defined as an effect of *transition* may also have actions assigned to it. Likewise, *behavior* can be defined for *state* as an *entry, exit*, and *doActivity*. The action is placed in an activity as an executable node which is a subtype of the *ActivityNode*. According to the subset of UML 2.5 metamodel in Fig. 1, the activity can be used to define behavior which therefore can either be used as a super behavior or be nested in some other behavioral model such as state machines. For example, the activity can be associated with the transition to define the effect of it as discussed in Section 2.3. However, in this work, we focus on the use of activities although various views are considered.

The *Activity* can be viewed to collectively represent the behavior of the atomic model or some of its constituents thereof. We ignore the holistic view of activities for the whole atomic model since that imposes some of the ordering relationships among the different functions which must be handled by the simulation protocol. Thus, we consider each function separately by devising an activity for each one, and devise activities for the subordinates thereof. The advantage of creating activities for the subordinates is to allow them to be composed in various ways with potentially different kind of models. For example, they can be contained within other activities as an action for external behavior or within state machines as an effect for a transition.

Table 1
Activity specifications for atomic DEVS model.

	A template for an activity diagram for the external transition function receiving a collection of inputs and processing them in iterative expansion region		
	A template for an activity diagram for the output function with an output parameter node		
	A template for an activity diagram for the internal transition function		An action to call behavior specified in another activity
	Read structural feature action to read the input and assign it to the output pin		Receive a value and assign it to the phase
	Read structural feature action to read the phase and assign it to the output pin		Set value and assign it to the output pin
	Read value of the received feature via the input pin		An action for making content to be used for preparing output

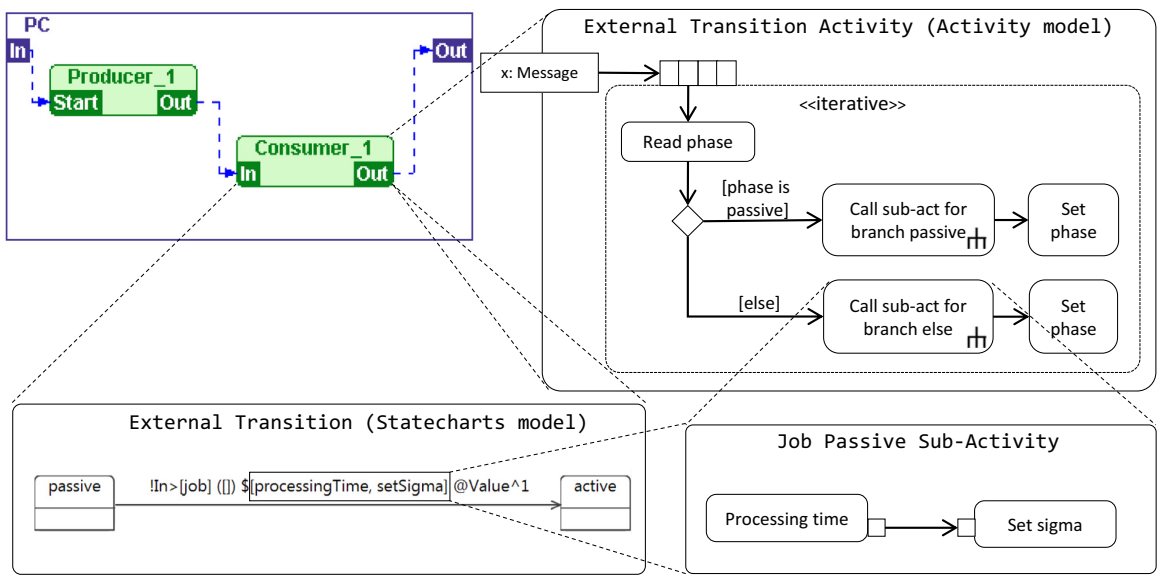


Fig. 2. The overall view of the approach and relationships between different models of consumer behavior.

We now create multiple activities for atomic model specified behavior by each function. These activities are created for modeling the behavior of an atomic DEVS model with multiple inputs. We model the external, internal transition and output functions, each in a separate activity. Activity models for the time advance and confluent transition functions can be specified in a similar manner. We use the modeling elements as described in Table 1 to provide the required modeling capabilities for specifying the behaviors of atomic DEVS models. The semantics of these elements as defined in the UML specification have been aligned with the DEVS concepts.

The activity diagrams shown in Fig. 2 depicts some of the behavioral specification of the processor model (a richer formulation to be demonstrated in Section 6) as well as a holistic view of the devised approach. The activity in the upper right corner depicts the behavior of the external transition function, while the activity in the lower right corner represent a subordinate unit for a state transition, therefore we call them sub-activities. A modeling engine specialized for developing these activity models is developed and will be discussed in the following section. A coupled Producer-Consumer (PC) model (upper left corner) and the external transition function belonging to the statecharts of the processor model (lower left corner) are developed in CoSMoS [12].

4.1. Categorizing the activity specification

According to the proposed specification (see Table 1), the components in activity-based models can be categorized into composite and primitive. The specifications for the external, internal transition, and output functions are composite. In every external transition function, a bag of external inputs is received and modeled as an activity parameter. The output function sends out the output which is modeled as an activity parameter although it is an output parameter. The internal transition function can manipulate the state of the model via use of some primitive components. These abstractions thereafter can serve as a basis for the behavioral specification of DEVS atomic model. And their contained components, whether composite or primitive, can collectively form the behavior of each individual function. Hence, the primitive components can also be used in different function or possibly different models such as the action of making content or setting the phase to passive. Therefore, the single action can also collectively form the behavior of the system while maintaining modularity.

The primitive components can be also characterized by their roles in the model such as changing the state from DEVS standpoint. Other primitive components may include any action that contributes to the behavior of the model such as assignment. Primitive components are the fundamental units of the behavior. Their semantics vary according to their role in the model. In changing the state, the model should essentially determine changes on the phase and the sigma as primary variables for the total state in the atomic model. In some assignment action, the value should be assigned to its corresponding variable.

The two categories provide a means to establish a stronger context especially for the primitive components. Since the objective of this work is to provide means for restraining complexity from a behavioral standpoint, the relationship between these components within some context is quite crucial. For example, the activity node is placed within an activity that is created to represent an external transition function. An investigation of whether the same node is simultaneously used in another activity for another element or not could be useful.

4.2. Note on coupled models and behavioral specification

It is important to note that the coupled model has a significant role in delivering the expected outcome of this research. For this reason, at the implementation level, we attempt to put things in a consistent perspective with the existing CoSMoS in which the specification of coupling is supported. CoSMoS recently has been equipped with the capability to behavioral modeling which is currently based on statecharts as mentioned previously in Section 2.3. However, the specification of an atomic model is encapsulated with respect to other atomic models. The behavior of a model can only be communicated through I/O ports and couplings with other models. Therefore, their abstractions can also differ in a way that preserves this property. An atomic model specified in some form can be coupled with some other atomic model specified in a different form. In our case, it should be possible to couple models specified in CoSMoS with their behavior modeled using statecharts. And, it should be also possible to couple them with atomic models in which their behavior is modeled using the activity-based specification as discussed in this work. This is important to help in achieving better outcomes toward the enrichment of the behavioral specification of the parallel DEVS formalism.

Moreover, the mandated behavior via coupling is also crucial in constituting the totality of system dynamics. Since the interaction between different atomic and coupled models is strictly via I/O ports and couplings, therefore, their influence on the inner atomic model logic is also restricted to the arrival of the external inputs. And their influence on other models is also contingent on sending outputs through their output ports.

The aforementioned concepts and capabilities are supported with many others in CoSMoS and integrated with DEVS-Suite. The creation of Parallel DEVS and Cellular Automata is supported in CoSMoS as component-based models [30]. Then, the corresponding code is automatically generated although for only the structural aspects. There is no code generation provided for statecharts modeling. This research contributes in providing richer modeling basis for further code generation and transformations. The path toward having a more disciplined simulation model development requires a collective achievement by multiple abstractions with further emphasize on their complementary role consistently.

4.3. Controlled coupling using activities control nodes

The provided capabilities in control nodes are used to control the transmission of input/outputs among different atomic and coupled models. Transmission of events (i.e, input to input, output to input, and output to output) among all components of any coupled, hierarchical DEVS model are instantaneous. This is described at a high level of system specification to represent the component interactions with each other via I/O ports thereof. The activity abstraction can become useful with respect to further mandating these interactions among different components. Various control capabilities of the flow can be used to describe the composition of components. The components themselves are modeled separately at a different level where the behavioral specifications can be described by activity specification as we discuss in this work or by different means such as statecharts. In the following, we discuss two common examples of handling the flow among different components.

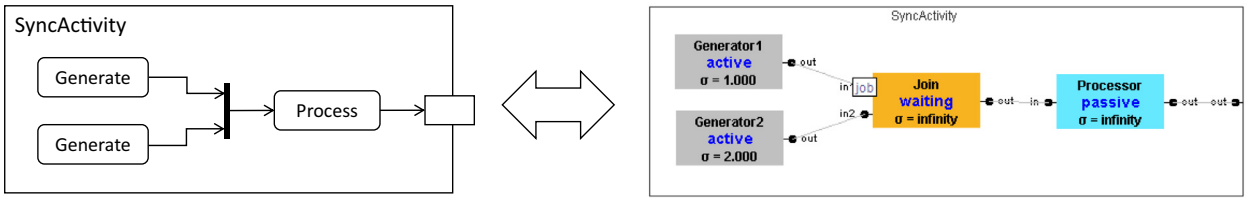


Fig. 3. An activity for synchronizing outputs from the generators prior to processing. The simulation view (right) is for the corresponding implementation in DEVS-Suite. The join node as an atomic model is in *waiting* phase to synchronize the input through the other port from the second generator.

4.3.1. Synchronizing inputs

As discussed, multiple inputs can arrive at some input ports and at arbitrary time instances at the DEVS atomic and coupled models. The join node can be used to synchronize multiple inputs being sent among different couplings. For example, the join node can be used to associate two different flows prior to an adder model. Consequently, the inputs will arrive at the same time instance at the adder input port. Hence, behavior of synchronizing multiple inputs has been separated from the domain model. Therefore, the domain model may or may not account for this need in its behavioral specification.

The join node can be used to specify synchronization of inputs as shown in Fig. 3. There are two generators that produce inputs for a processor model. The outputs from both generators are held at some point prior to their arrival at the processor model. This can be specified as a join node which is modeled as an atomic model with its behavior to be specified to encompass the semantics of the join node. The outputs by generators do not have to be necessarily produced at the same time instant. However, having the synchronization ensures the simultaneous arrival of the inputs at the processor's input port. This is defined in the behavior of the corresponding atomic model to the join node where the model has multiple input ports. Upon arrival of an input, the phase is changed to *"waiting"* to denote the existence of an input and therefore waits for other inputs from other input ports to arrive. Upon the arrival of all the inputs through, the output is dispatched. The phase is changed then to *passive* by the internal transition function. The formal specification of the atomic model for the join is defined in Parallel DEVS as

$JOIN = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$

where

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values

$S = phase \times \sigma \times condition1 \times condition2 \times store$, *where*

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$phase = \{passive, waiting\}$, $\sigma = \mathbb{R}^{+}_{0,\infty}$,

$condition1 = \{true, false\}$, $condition2 = \{true, false\}$, $store \in X_P$

$IPorts = \{in1, in2\}$, *where* $X_p = V$ (an arbitrary set)

$OPorts = \{out\}$, *where* $Y_p = V$ (an arbitrary set)

$\delta_{int}(phase, \sigma, condition1, condition2, store) = (passive, \infty, false, false, x)$ *where* $x \in X_P$

$\delta_{ext}((phase, \sigma, condition1, condition2, store), e, X_M) =$

$((waiting, \infty, true, condition2, (p_i, v_i))$ *if* $p_i = in_1$ *and* $condition2 = false$

$((waiting, \infty, condition1, true, (p_i, v_i))$ *if* $p_i = in_2$ *and* $condition1 = false$

$((waiting, 0, condition1, condition2, (p_i, v_i))$ *if* $p_i = in_1$ *and* $condition2 = true$

$((waiting, 0, condition1, condition2, (p_i, v_i))$ *if* $p_i = in_2$ *and* $condition1 = true$

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(waiting, \sigma, store) = (out, store)$

$ta(phase, \sigma) = \sigma$.

The atomic model is defined with two input ports and one output port. The model is initially in the phase *"passive"* until some input is received through any port. There are two conditions that are defined as secondary state variables to recognize the port through which the input arrives. Both conditions are initialized with false. As soon as an input is received, the external transition function toggles the condition corresponding to the input port. For example, if the input is received through port *"in1"*, then the toggled condition is *"condition1"*, and so on. If all conditions are true, the sigma is assigned a zero value causing the output function to occur. The output function then sends out all the stored received inputs. Finally, all state variables are set to their initial values in the internal transition function.

4.3.2. Network switch

Just as we can model the synchronization as a join node, we can also model the network switch by which some decision ought to be made in order to direct flow of input and output events. The synchronization is not necessary though. The selected control element is the decision node in terms of activity modeling. The switch directs the flow by sending the received inputs through the corresponding output ports under some defined condition.

As discussed in [31], the switch model illustrates the handling of multiple inputs arriving simultaneously as in Parallel DEVS. In that model, the switch is characterized to be a controlling component while the processor model exhibits a differ-

ent behavior. Together, they constitute, along with the other elements, the behavior of the network switch coupled model. The behavior of the processor model is described in details in Section 6.

5. EMF-based modeling engine

It is important to assist modelers in developing specifications for all parts of any atomic model. To achieve this objective, the approach proposed in this paper requires building a modeling engine that supports the activity-based modeling concepts and constructs introduced in Section 2.4. This engine should be robust and powerful enough to handle the potential complexity that might arise in DEVS activity behavior modeling (i.e., from individual to composite parts forming atomic functions). Therefore, we expand our work by constructing a modeling engine that utilizes the previous specifications and translate them into a runtime environment. The resulting engine accounts for formal specifications of DEVS as well as concepts introduced for atomic model behavioral specification. It also leverages the available tools and technologies in order to facilitate the process of building those models as well as ensuring the extensibility to allow for future works including model verification and simulation validation.

5.1. Activity-based DEVS Ecore

Using the current realization of the UML2 metamodel in EMF [32], Activity metamodel is realized in the same manner to provide the necessary support for DEVS atomic model behavioral specification. The created Ecore is currently aimed to be incorporated with EMF-DEVS and used to provide a graphical modeling capability using Eclipse Graphical Modeling Framework (GMF). The Ecore consists mainly of activities and DEVS packages as shown in Fig. 4. DEVS package contains basic elements from EMF-DEVS for both structure and behavior as discussed in previous works [6,8] with some additional elements. The activities package contains elements to realize the activity-based specification. They have been created by taking into account the UML 2.5 specification as well as UML2 metamodel [32] in Ecore.

We construct the relationships between DEVS, activities, and actions metamodels using extensions and references via ESuper types and EReference as defined in Ecore [19]. The state transition functions, output, and time advance functions are specialized from DEVSActivity, that is, specialized from the *Activity* EClass. The goal is to maintain common properties among those four functions in this common abstract DEVSActivity thus they become distinguishable from other activities. References are established in the Atomic EClass for each DEVS function. The atomic model may have up to one of δ_{ext}, δ_{int}, δ_{con}, and λ functions. For example, the reference for the external transition function may not be set for the producer model. Therefore, the reference is defined to have zero as a lower bound and one as an upper bound. The references are contained within their corresponding atomic model. This is to ensure that the strong modularity required for the DEVS formalism is maintained.

Specializing actions for DEVS modeling is accomplished in a similar manner. The behavior can be then characterized by those actions in addition to the comprehensive set of actions defined in UML. The current list of actions as specialized in the Ecore includes (but not limited to) the following:

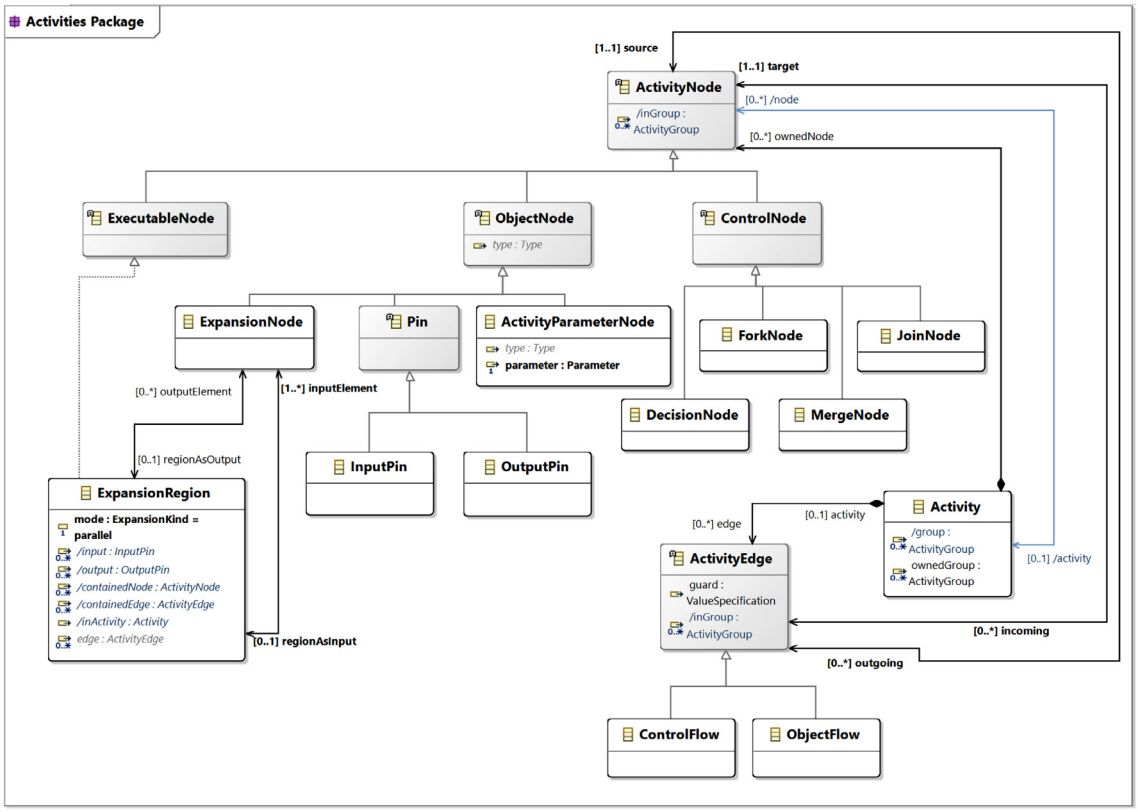
- *SetState*: to change the current state due to some state transition. This component is defined to be abstract.
- *ReadState*: the current state is checked before making changes to some state. This component is defined to be abstract.
- *ReadValueOnPort*: to read input *value* that has arrived via some input *port*.
- *SetSigma*: to assign a time period for some state.
- *MakeContent*: create *(port,value)* pairs where one or more output events are assigned to one or more output ports.

This set of actions are based on the commonly used in many DEVS atomic models supported in the DEVS-Suite simulator. However, they are generalized and thus can be employed by other simulators that conform to the DEVS formalism. Their implementations can be specified and mapped according to their definitions in the metamodel.

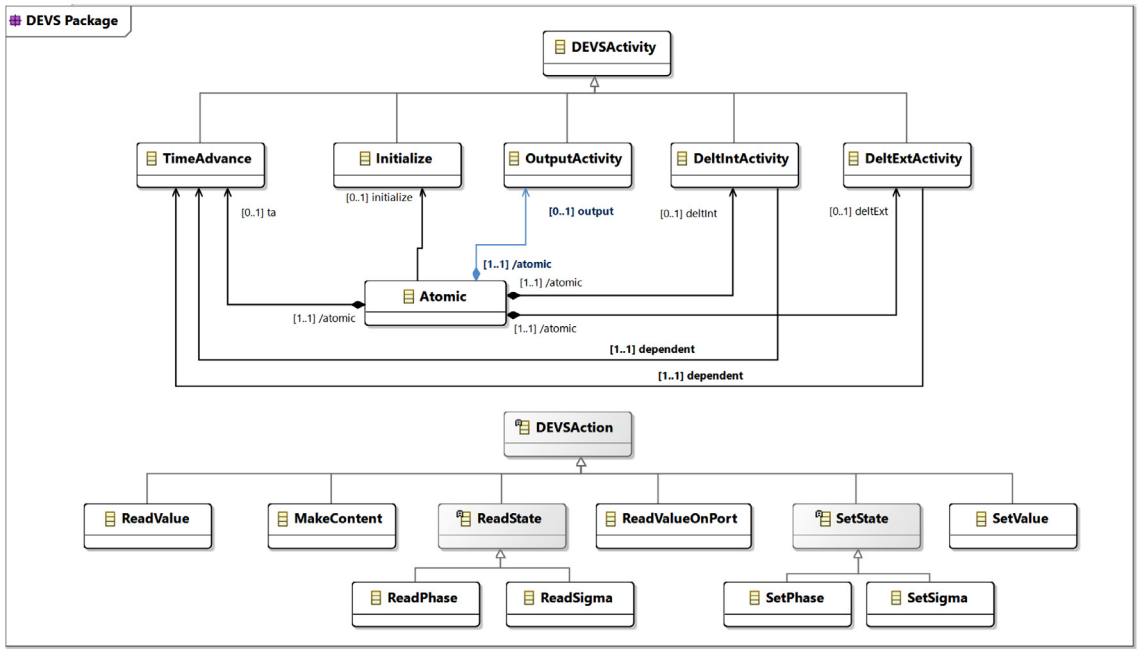
5.2. Activities graphical definition and tooling

The GMF framework supports creating visual editors for modeling languages such as statecharts. We have leveraged this framework and built an expressive graphical notation for modeling behavior based on the activity-based approach. The visual syntax is defined to be consistent with that of the UML activity modeling. Therefore, any activity model created using the activity graphical definition is a legitimate UML activity diagram. The specialized elements for DEVS are given their superclass graphical definitions. For example, *SetPhase* action is defined to have same visual notation as the one for manipulating the structural feature that it specializes. This modeling engine supports development of Activity-based DEVS models such as those shown in Fig. 2

The canvas for Activity-based DEVS allows creating behavior models through use of nodes and connectors. The canvas consists of figure galleries, nodes, connections, compartments, and labels. The figure gallery is used to define a set of graphical notations enabling defining the elements of any atomic models as depicted in Fig. 5. GMF by default generates rectangles for node elements and solid polylines for link elements. The definition is modified to represent the graphical notation for activities. The notation includes a variety of unique shapes such as rectangles, rounded rectangles, arrows, and diamond. Customization is often required for more specific shapes. A sample of the graphical definition for the decision



(a) An Ecore package containing Activities metamodel



(b) An Ecore package containing the behavioral constituents of DEVS metamodel and actions

Fig. 4. Activity-based DEVS metamodel.

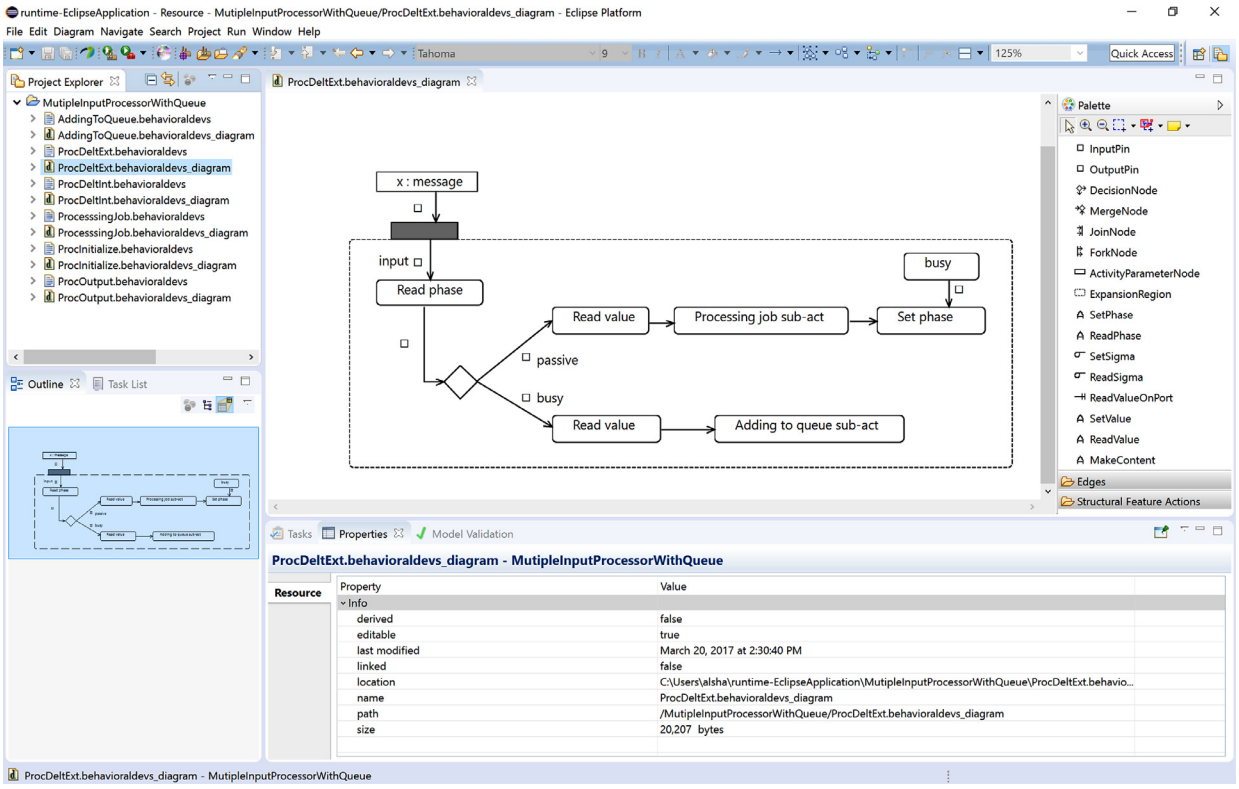


Fig. 5. Visual canvas for Activity-based DEVS modeling.

Table 2
Decision and merge node figure definition.

Element	Property	Value
Canvas	Name	BehavioralDEVS
Figure Gallery	Name	Activities Figures
Figure Descriptor	Name	Diamond
Rectangle	Name	Diamond
	Fill	False
	Outline	Flase
Polygon	Name	Diamond
Template Point	X, Y	15,0
Template Point	X, Y	0,15
Template Point	X, Y	15,30
Template Point	X, Y	30,15
Node	Name	DecisionNode
	Figure	Diamond
Default Size Facet		
Dimension	Dx, Dy	30,30
Node	Name	MergeNode
	Figure	Diamond
Default Size Facet		
Dimension	Dx, Dy	30,30

node is shown in Table 2. Graphical definitions for all other elements have been created in a similar manner. Note that we have made minor changes on some notations for simplicity reasons. One change is on the expansion node notation by making it a grey rectangle instead of a divided rectangle. The other change, we do not include the rake symbol on the call behavior action. Instead, the action name is appended with *sub-act* keyword at the end.

The tooling definition is created to define the used tools for creating the graphical diagrams such as palette and popup menus. Those tools can be then customized further to support specialized graphical interfaces to facilitate the modeling process – this means primitive and composite model elements are ensured to satisfy the combined UML Activity and DEVS behavioral model development. This kind of tooling allows organizing the modeling elements in more useful ways for the modelers and therefore enabling easier and simpler modeling. The UI includes, but not limited to, features like palette

grouping, icon images, modeling hints, and guidelines. With these features modelers are well-positioned to specify complex behaviors incrementally and iteratively. The palette elements are organized and separated according to their classification from a semantic point of view for both activities and DEVS modeling. The palette lists UML activity nodes followed by the DEVS nodes such as the nodes for specifying phase and sigma. Specialized image icons for nodes are provided to capture their mathematical counterparts in the DEVS formalism. For example, *SetSigma* is associated with an image icon of sigma symbol (σ).

5.3. Mapping

Finally, we create a mapping model where the elements defined in metamodel are mapped to their corresponding counterparts in the graphical definitions and tooling. This leads to combining the three essential models in GMF (i.e., the domain model which is in our case the metamodel, the graphical definition, and the tooling definition). GMF wizard allows to distinguish between nodes and links in the mapping model. However, it only allows for basic customization capabilities. The tooling model is manipulated further. The model, graphical, and tooling elements are linked with each other using the node mapping feature and specifying its properties accordingly.

5.4. Preliminaries on the validation of the activity-based DEVS models

The validation of the proposed approach is currently based on the provided validation capabilities by EMF. The metamodel as shown in Fig. 4 captures the concepts by representing them in a set of classes, attributes, and references. It thereafter forms the ground for further constituting the validity of the instantiated models. After being specified, the validation is performed by the EMF engine.

Although a thorough validation study is considered as a future work, we briefly discuss some of the validation capabilities and benefits obtained with respect to the behavioral specification. Many of these benefits are acquired through the merit of the existing metamodel itself in general. In addition, many others are also acquired by the virtue of effectively utilizing the underlying framework. EMF engine is equipped with a validation engine that allows defining and checking validity of model constructs. Any validation can be conducted by simply customizing the default properties of different models as far as going all the way to modifying the generated code. Defining more constraints yields possibly to a more restrictive modeling environment. Therefore, the legitimacy of the constraint must be ensured to be valid for all corresponding models before enforcement. For example, the value of the sigma must be always non-negative. This constraint is defined and enforced with an error message in case of violation. Less restrictive constraints can also be defined. In the case of not receiving any inputs, a warning message is only shown because in some special cases the model does not have to receive inputs such as a generator model that is autonomous (i.e. has no input ports).

For any non-mentioned validation whether it is a general property or specific to a certain domain, the existence of the EMF-based metamodel provides a suitable ground nonetheless to simplify the process of adding further analyses by defining constraints or through extension mechanism. This can serve for behavioral specification of domain specific models (e.g., [33]). A framework is proposed to deal in part with complications that is likely to be encountered during the model development with respect to its behavior. Linking to such domains can be possible after establishing the necessary extensions and transformations.

Although some constraints can be validated to ensure the legitimacy of model specification, validation in the sense of simulation is impractical due to the infinite state space of DEVS models. Further constraints can be made for a specific domain on the model inputs, outputs, and states, for validation purposes. The constraints for models complement the validation of simulated behavior. They can be made on the state transitions by controlling the elements that deals with state changes (e.g., order in which state or non-state variables can change their values). Extracting inputs can be also checked in the reading inputs and ports elements in conjunction with their corresponding sets as defined by the model. The same can be also performed for outputs. This has been made possible after dissecting into the behavioral modeling approaches by defining, categorizing, and characterizing each individual element and looking into its specifics thereafter.

6. Activity-based modeling for multiple input processor with queue

A model of a processor with a buffer exemplifies the essence of the Parallel DEVS formalism. The processor can handle multiple input events (e.g., jobs to be processed) from the standpoint of their arrival and storage. Multiple inputs may arrive simultaneously on one or more input ports. The buffer is used to store jobs when the processor is busy in processing another job. The stored jobs are processed in the order in which they have stored in the queue. The model is defined in Parallel DEVS [1] as

$Processor_{queue} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$

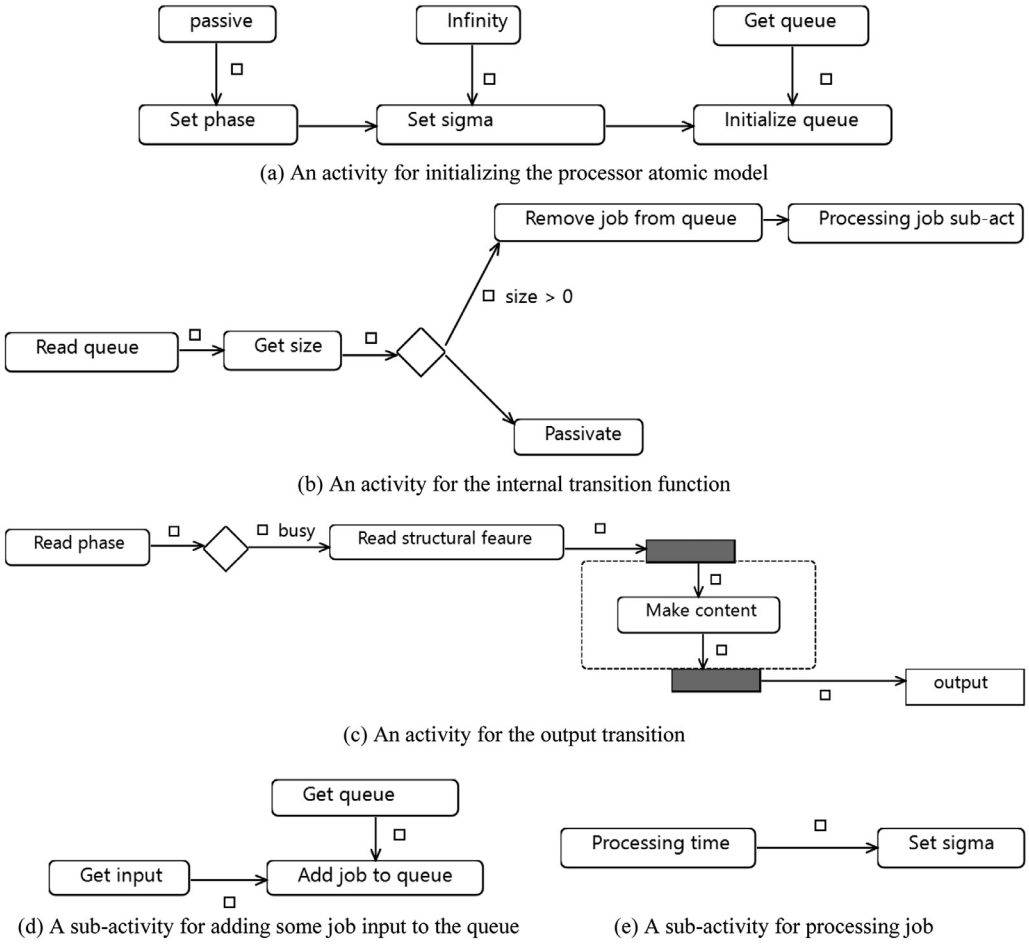
where

$IPorts = \{in\}, where \quad X_{in} = V_X$ (an arbitrary set)

$X = \{(p, v) | p \in IPorts, v \in X_{in}\}$ is the set of input ports and values

$S = \{passive, busy\} \times \mathbb{R}_0^{+\infty} \times q$

$OPorts = \{out\}, where \quad Y_{out} = V_Y$ (an arbitrary set)



(a) An activity for initializing the processor atomic model

(b) An activity for the internal transition function

(c) An activity for the output transition

(d) A sub-activity for adding some job input to the queue

(e) A sub-activity for processing job

Fig. 6. Activity-based DEVS modeling for multiple input processor with queue.

$Y = \{(p, v) | p \in OPorts, v \in Y_{out}\}$ is the set of output ports and values

$\delta_{int}(phase,\ \sigma,\ q)$

$= (passive, \infty, q)$ if queue is empty

$= (busy, processingTime, q')$ otherwise remove head of the queue

$\delta_{ext}((phase,\ \sigma,\ q),\ e,\ ((\text{“}in\text{”},\ x_1),\ (\text{“}in\text{”},\ x_2),\ \ldots,\ (\text{“}in\text{”},\ x_n)))$, $x_i \in X_{in}$

$= ((busy, processingTime), x_1, x_2, \ldots, x_n)$ if phase $= passive$

$= ((phase, \sigma - e), q.(x_1, x_2, \ldots, x_n))$ otherwise add input events to the tail of the queue

$\delta_{con}((s, ta(s)), x) = \delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(phase, \sigma, q) = (out, q.head)$, $q.head \in Y_{out}$, if phase $= busy$

$ta(phase, \sigma, q) = \sigma$.

Activity models are created for the internal transition, external transition, and output functions (see Fig. 6). Specifying the remaining function is straightforward from activity modeling standpoint. Modeling some parts of the processor model (e.g., queue) are currently out of our scope; however they are being manipulated in which their impact cascades to the specified behavior. We note that the initialization and termination of the modeled activities are not shown. This is due to the fact that the simulation protocol is responsible for handling the execution of the different functions including their orders in a way that complies with the operational semantics of the abstract simulator.

A major advantage of the developed modeling engine is that it enables the modelers to visualize a wider array of DEVS activity models. It enables viewing the model in various ways including textual representation although this representation was not our main focus. The powerful unifying capability attained by the reliance on Ecore reduces some difficulties associated with modeling approaches including the UML state machine.

```
public void  deltext(double e,message x)
{
  Continue(e);

  for (int i=0; i< x.getLength();i++){
    if (phaseIs("passive"))
      for (String inPort : inPorts)
        if (messageOnPort(x,inPort,i))
        {
          job = x.getValOnPort(inPort,i);
          sigma = INFINITY;
          phase = "busy";
        }
    else if (phaseIs("busy"))
      for (String inPort : inPorts)
        if (messageOnPort(x,inPort,i))
        {
          job = x.getValOnPort(inPort,i);
          q.add(job);
        }
  }
}
```

Listing 1. External transition function for the processor with queue.

6.1. Interpreting the processor model in the DEVS-Suite simulator

The path toward concrete realizations of the proposed abstraction can be accomplished through model transformation. More elements have to be accounted for since the concrete models are more restrictive than their representations at some higher level. Concepts such as visibility and data typing need to be strictly defined by the transformation in order to achieve fully specified models. In the case of the DEVS-Suite simulator, the code that corresponds to the atomic model can be achieved via interpretation or code generation. The manipulation of the primary state variables, which are the phase and sigma, is easier than the secondary ones especially with specific parts such as queue. Queue is classified as a non-simulatable model [30] as compared with the simulatable processor model. Therefore, since our focus is the behavioral specification, we assume that the queue is already defined as in [8] and as a result the manipulation of its elements could be performed directly during the transformation. The implementation of the external transition function is shown in Listing 1. The *for* loop is for checking any bag of received inputs. Then, the phase is checked as in the *if* statement in correspondence to the decision node in the activity model. The phase is defined as a *String* in the DEVS-Suite simulator which is considered in the transformation, not in the activity model. The sigma is also manipulated in a similar manner while the data type is *double*. The details are accounted for during the transformation to complement models at the higher level with the necessary constructs and definitions to make them concrete.

6.2. Generating simulatable implementations

The Model Driven Architecture (MDA) [34] is a semi-formal methodology with the aim to allow systems to be conceptualized at four different layers. Unlike other methodologies, MDA establishes a four-layer hierarchy starting from the highest layer M3 to the lowest layer M0. For each of the M3 and M0 as well as the intermediary M2 and M1 meta-layers, an arbitrary number of layers may be defined. The aim of this four-layered architecture is to generate code from modeling diagrams specified by developers. Different languages (e.g., Structured Classifiers) can be identified with respect to the MDA layers and thus collectively used for model specification. For example, modelers can use both activities and statecharts to define complementary behavioral aspects of their models. This is in fact one of the major goals of MDA as long as these

representations have their metamodels based on the Meta-Object Facility (MOF) [35]. The architecture captures languages at some level in the hierarchy and enable generative technologies [36, chap 9] toward higher level goals such as ensuring architecture-to-implementation consistency.

As opposed to architectures without having predefined layers and transformation relationships, MOF is defined to strictly bound the M2, M1, and M0 meta-layers at the M3 level. Thus, MOF models at layer 3 are assumed to be capable of defining themselves as well as constraining models at lower meta-layers. Models at the M2 layer conform to MOF and can be called metamodels. Objects are identified as in object-oriented paradigm, however, they are used as metaobjects without revealing so much about their specifics. This serves as a basic step toward achieving the development of a platform-independent models where models are specified without being tied to any specific platform and implementation. Models at the M0 layer are the instances for an actual target software system.

Although MDA has a fundamental role in taming model structure and behavior via layers, it is not easy to have implementations or realizations that properly manifest models at higher meta-layers. In fact, it is "extremely difficult" [36] to have complete implementations of system elements. Although simulation is akin to models that lends themselves to formalisms such as DEVS, the purpose of the models is not solely to generate code. Therefore, implementations of models can be acquired only to some extent in the context of the generative technologies for modeling approaches in which MDA concepts are accounted for.

Our approach recognizes the architecture-to-implementation challenge and yet undertakes an ambitious attempt toward further partial implementations for simulation models while enabling disciplined (Model-Driven Engineering) behavioral modeling grounded in formal modeling theories. Any such implementation is the result of combining a mixture of component, activity, and statecharts specifications enabled by modeling theories (e.g., parallel DEVS) and realized using MDA-based frameworks (e.g., GMF). We also realize the necessity of encoding specification in general or targeting specific simulators. However, we address the problem differently by distinguishing between code fragments as realizations of some system elements and their purposes in the final product. The code can be written in many different ways. It can be predefined for some components. It can be also implemented using transformations within well-defined meta-layers. Last but not least, it can be manually encoded. The generated code can also differ with respect to what it represents. Some code corresponds to structural specifications which is relatively simpler to achieve relative to the behavioral specifications. For example, in our previous work [30], the generated code fragments are for structural specification where further implementations are necessary for behavioral specification in order for them to be simulatable in DEVS-Suite.

To summarize, implementations incarnate in different elements within the MDA and our proposed framework. Implementations, that are achieved or considered to this point, can manifest themselves in the following elements:

- **The simulation protocol**: this is at the execution core for the DEVS-Suite simulator. All generated code fragments and transformations must comply with this protocol. We mention this element to clarify the implementation perspective in the context of the simulation. The approach is not restricted to any specific simulator as long as a chosen simulator complies with the given simulation protocol.
- **EMF and GMF code generation facilities**: these facilities are provided within both frameworks. We utilize them to facilitate the creation and validation of Activity-based DEVS models.
- **Transformation of activities in DEVS-Suite**: This is achieved as a transformation facility which is implemented as an activity interpreter for DEVS-Suite.
- **Transformations of activity nodes and actions in DEVS-Suite**: these transformations are achieved through a set of atomic models that are implemented by extending the viewable atomic class as defined in DEVS-Suite. Some atomic model corresponds to some control node such as *join* as discussed in Section 4.3.1.

7. Conclusion

Use of various modeling approaches and methodologies for behavioral modeling is invaluable as it paves the way for reasoning about the system's properties and dynamics. We aim to contribute to this need by providing a basis for this emerging approach for behavioral specification of discrete system modeling and specifically for the parallel DEVS formalism. We described the basis for Activity-based DEVS modeling. We considered different views for adopting activities as a candidate behavioral modeling language for specifying DEVS atomic model (and by extension coupled) behavior. The view has to account for compliance with concepts from formal and semi-formal modeling methodologies to enable and enrich utilization and enrichment of model development. We proposed an approach where behavioral abstractions in the DEVS formalism and statecharts are extended with those of the UML activity.

We chose MDA as the basis for defining foundational artifacts key for detailing behaviors belonging to the DEVS atomic model functions. The potential value of behavioral metamodeling concepts for developing simulation models can be unlocked by proper employment thereto among different MDA layers. As the complexity and scale of systems continue to grow, disciplined use of the DEVS modeling formalism and the model driven development, especially with respect to system behavior simulation, is attractive.

We demonstrated the approach by developing a first-generation Activity-based DEVS modeling engine supported with multiple capabilities using the GMF framework and tool. This engine facilitates developing atomic model behaviors at scale.

This GMF-based tool supports evolving model behavior with automation (e.g., ensuring conformance to the DEVS syntax and semantics for functions).

Promising future work includes furthering behavioral artifacts supporting higher-order control structures. These can be key for building more useful simulations for Systems of Systems [37] including Cyber-Physical Systems and Internet-of-Things. Another future research interest is to reduce the size of behavioral models in a manner similar to design and code refactoring.

Acknowledgment

We acknowledge helpful comments from anonymous referees on the earlier versions of this paper.

References

[1] B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.
[2] S. Kim, H.S. Sarjoughian, V. Elamvazhuthi, DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring, in: Proceedings of the 2009 Spring Simulation Multiconference, in: SpringSim '09, Society for Computer Simulation International, 2009, p. 161.
[3] R. Soley, the OMG Staff Strategy Group, Model driven architecture, OMG White Paper 308 (2000).
[4] J.L. Risco-Martin, J.M. de la Cruz, S. Mittal, B.P. Zeigler, eUDEVS: executable UML with DEVS theory of modeling and simulation, Simulation 85 (11-12) (2009) 750–777.
[5] D. Cetinkaya, A. Verbraeck, M.D. Seck, MDD4MS: a model driven development framework for modeling and simulation, in: Proceedings of the 2011 Summer Computer Simulation Conference, Society for Modeling & Simulation International, 2011, pp. 113–121.
[6] H.S. Sarjoughian, A.M. Markid, EMF-DEVS modeling, in: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium, Society for Computer Simulation International, 2012, pp. 1–8.
[7] S. Mittal, J.L.R. Martín, Netcentric System of Systems Engineering with DEVS Unified process, CRC Press, 2013.
[8] H.S. Sarjoughian, A. Alshareef, Y. Lei, Behavioral DEVS metamodeling, in: Proceedings of the 2015 Winter Simulation Conference, in: WSC 15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 2788–2799.
[9] OMG, Unified Modeling Language version 2.0, 2005.
[10] OMG, Semantics of a foundational subset for executable UML models (fUML), 2012. URL http://www.omg.org/spec/FUML/.
[11] A. Alshareef, H.S. Sarjoughian, B. Zarrin, An approach for activity-based DEVS model specification, in: Proceedings of the Symposium on Theory of Modeling & Simulation, Society for Computer Simulation International, 2016, p. 25.
[12] ACIMS, CoSMoSim, 2015. URL https://sourceforge.net/projects/cosmosim/.
[13] C. Chidisiuc, G.A. Wainer, CD++ Builder: an eclipse-based IDE for DEVS modeling, in: Proceedings of the 2007 spring simulation multiconference-Volume 2, Society for Computer Simulation International, 2007, pp. 235–240.
[14] F. Bergero, E. Kofman, PowerDEVS: a tool for hybrid system modeling and real-time simulation, Simulation 87 (2011) 113–132.
[15] D.A. Hollmann, M. Cristiá, C. Frydman, CML-DEVS: a specification language for DEVS conceptual models, Simul. Model. Pract. Theory 57 (2015) 100–117.
[16] B.P. Zeigler, H.S. Sarjoughian, Introduction to DEVS Modeling and Simulation With Java: Developing Component-Based Simulation Models, Arizona State University, 2003.
[17] D. Harel, M. Politi, Modeling Reactive Systems With Statecharts: the STATEMATE Approach, McGraw-Hill, Inc., 1998.
[18] OMG, Unified Modeling Language version 2.5, 2012.
[19] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[20] R.C. Gronback, Eclipse Modeling Project: A Domain-Specific Language (DSL) toolkit, Pearson Education, 2009.
[21] M.D. Fard, H.S. Sarjoughian, Visual and persistence behavior modeling for DEVS in CoSMoS, in: Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium, 2015, pp. 227–234.
[22] S. Schulz, T.C. Ewing, J.W. Rozenblit, Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling, in: Proceedings Seventh IEEE International Conference and Workshopon the Engineering of Computer Based Systems, 2000. (ECBS 2000), 2000, pp. 308–316.
[23] D. Zinoviev, Mapping DEVS models onto UML models, DEVS Symposium, Spring Simulation Multiconference, 2005.
[24] J. Mooney, H.S. Sarjoughian, A framework for executable UML models, in: Proceedings of the 2009 Spring Simulation Multiconference, Society for Computer Simulation International, 2009, pp. 1–8.
[25] H.S. Sarjoughian, S. Gholami, Action-level real-time DEVS modeling and simulation, Simulation 91 (10) (2015) 869–887.
[26] M. Nikolaidou, V. Dalakas, L. Mitsi, G.D. Kapos, D. Anagnostopoulos, A SysML profile for classical DEVS simulators, in: Proceedings of the 2008 The Third International Conference on Software Engineering Advances, 2008, pp. 445–450.
[27] D. Foures, V. Albert, J.C. Pascal, A. Nketsa, Automation of SysML activity diagram simulation with model-driven engineering approach, in: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium, 2012, pp. 1–6.
[28] O. Ozmen, J. Nutaro, Activity diagrams for DEVS models: a case study modeling health care behavior (WIP), in: Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium, 2015, pp. 1006–1011.
[29] L. Yonglin, W. Weiping, L. Qun, Z. Yifan, A transformation model from DEVS to SMP2 based on MDA, Simul. Model. Pract. Theory 17 (2009) 1690–1709.
[30] H.S. Sarjoughian, V. Elamvazhuthi, CoSMoS: a visual environment for component-based modeling, experimental design, and simulation, in: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, p. 59.
[31] A. Alshareef, H.S. Sarjoughian, DEVS specification for modeling and simulation of the UML activities, in: Proceedings of the Symposium on Model–driven Approaches for Simulation Engineering, Society for Computer Simulation International, 2017, p. 9.
[32] Eclipse Foundation, Model development tools (MDT), 2016. URL http://www.eclipse.org/modeling/mdt/.
[33] Z. Zhu, Y. Lei, Y. Zhu, A. Alshareef, H. Sarjoughian, A unifying framework for UML profile-based cognitive modeling: development and experience, in: Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques, Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 2017.
[34] J. Miller, J. Mukerji, MDA guide version 1.0. 1, Object Management Group, 2003, (http://www.omg.org/docs/omg/03-06-01).
[35] OMG, Meta Object Facility version 2.5.1, 2016.
[36] R.N. Taylor, N. Medvidovic, E.M. Dashofy, Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2010.
[37] J.W. Marvin, J.T. Schmitz, R.A. Reed, Modeling-simulation-analysis-looping: 21st century game changer, in: INCOSE International Symposium, 26, Wiley Online Library, 2016, pp. 803–816.