*Proceedings of the 2018 Winter Simulation Conference*
*M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.*

# A SYMMETRIC FORMALISM FOR DISCRETE EVENT SIMULATION WITH AGENTS

Rhys Goldstein
Simon Breslav
Azam Khan

Autodesk Research
661 University Avenue, Suite 200
Toronto, ON M5G 1M1, CANADA

## ABSTRACT

In designing a general modeling formalism for domain experts, a key challenge is to support a broad selection of their preferred paradigms yet minimize their exposure to complexity. With this aim, a formalism called Symmetric DEVS is proposed for specifying models that incorporate elements of discrete event simulation, dataflow programming, and agent-based modeling. Symmetric DEVS is based on the Discrete Event System Specification (DEVS) formalism, but differs in that atomic and composite nodes for discrete events are complemented with function and collection nodes for dataflow and agents. Like DEVS, nodes communicate over simulated time via message ports, but they also feature flow ports accommodating initialization and finalization operations. To minimize conceptual complexity, specifications are pared down to the essential elements and formulated to exhibit a high degree of symmetry. This paper defines the mathematical elements of Symmetric DEVS and presents an example of each of the four types of nodes.

## 1 INTRODUCTION

The appeal of the Discrete Event System Specification (DEVS) lies in its ability to represent almost any time-dependent behavior (Vangheluwe 2000), as well as its hierarchical approach to model development. It is worth noting, however, that certain types of model hierarchies pose challenges for the original formalism. In particular, Classic DEVS does not support a type of composition ubiquitous in agent-based modeling (ABM), where the number of instances of an agent model is arbitrary and permitted to vary over the course of a simulation. A number of dynamic structure DEVS variants have therefore emerged which allow changes to a model hierarchy to unfold over simulated time. Another concern is how to make DEVS approachable to domain experts, who may perceive the formalism as unfamiliar and complex.

Although dynamic structure variants of DEVS provide comprehensive support for ABM and other simulation techniques, they unfortunately introduce a significant number of additional elements into model specifications. The added complexity may further discourage adoption by domain experts. This motivates the search for a middle ground between Classic DEVS and its full-fledged dynamic structure variants, a solution that accommodates ABM while minimizing conceptual complexity. Because agents may need to be created and terminated during a simulation, one implication of ABM is that the initialization and finalization of model instances becomes a key concern. To handle these operations, our proposal incorporates a paradigm that is gaining traction among end-user programmers in a variety of fields: dataflow programming. Dataflow nodes are symmetric in form, a quality we attempt to infuse into DEVS-inspired nodes.

In this paper we introduce Symmetric DEVS, a set of conventions designed to yield minimal model specifications while supporting discrete event simulation, dataflow programming, and agent-based modeling. Symmetric DEVS models are defined by formally specifying and linking nodes of four types: atomic nodes, which are similar to Classic DEVS atomic models; function nodes, which we borrow from dataflow programming; composite nodes, which combine Classic DEVS coupled models with dataflow; and collection

nodes, which we introduce to accommodate arbitrary numbers of agents of the same type. We have observed that in many virtual experiments, a considerable amount of code is dedicated to pre-computations that help initialize simulation models. Furthermore, there are usually operations to perform at the conclusion of a simulation run. Symmetric DEVS nodes include flow ports to aid in the specification of these initialization and finalization procedures. Atomic, composite, and collection nodes also retain the message ports of Classic DEVS. The four types of ports—flow input ($P_{\text{fi}}$), message input ($P_{\text{mi}}$), message output ($P_{\text{mo}}$), and flow output ($P_{\text{fo}}$)—exhibit a symmetry that is intended to make the formalism more approachable and easier to recall. Symmetry is also emphasized in the names and symbols of the four main event handling functions ($f_{\text{init}}$, $f_{\text{u}}$, $f_{\text{p}}$, $f_{\text{final}}$), the two types of durations ($\Delta t_{\text{e}}$, $\Delta t_{\text{p}}$), and the macro and micro levels ($\Psi_{\text{M}}$, $\Psi_{\mu}$) of composite and collection nodes.

Symmetric DEVS is designed to serve not as a rigid standard for model specification, but rather as an option for modelers who wish to formally express agent creation/termination and/or initialization/finalization operations in conjunction with discrete event behavior. Our philosophy is that the formalism should be extended as needed, and thus we present only the most essential elements. We exclude state sets, for example, which are implicit in the Symmetric DEVS event handling functions. We also exclude the Classic DEVS tie-breaking function, proposing instead a non-deterministic interpretation of simultaneous event ordering. If desired, we encourage modelers to expand their specifications by adding state sets, tie-breaking functions, and other elements on a per-application basis.

This paper (*a*) describes the main concepts underlying Symmetric DEVS and their relationship to prior work, (*b*) defines the mathematical elements of the formalism, and (*c*) presents an example of each of the four types of nodes. The examples have been tested using the SyDEVS library, an open source implementation of Symmetric DEVS developed in C++.

## 2 BACKGROUND AND RELATED WORK

### 2.1 DEVS and Approachability

The Classic DEVS formalism was invented in the 1970s to allow the behavior of essentially any time-varying system to be represented in a common form. Systems are formally described as atomic or coupled models containing the elements in (1), where $X$, $Y$, $S$, $D$, $EIC$, $EOC$, and $IC$ are mathematical sets, $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, $ta$, and $Select$ are functions, and $M_d$ is a DEVS model. Zeigler et al. (2000) provide the complete definition.

$$\begin{array}{l|l} \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, ta \rangle & \text{Classic DEVS atomic model} \\ \langle X, Y, D, \{M_d\}, EIC, EOC, IC, Select \rangle & \text{Classic DEVS coupled model} \end{array} \tag{1}$$

Although DEVS offers domain experts a scalable approach for model development, its nomenclature and intentional de-emphasis of the simulation procedure are unfamiliar to most programmers. Hence numerous efforts have been undertaken to make DEVS more approachable and encourage its adoption. Such efforts involve the development of visual modeling interfaces, particularly 2D network editing interfaces for coupled models as seen in GVLE (Quesnel et al. 2009) and many other graphical DEVS-based simulation environments. Some packages, such as CD++Builder (Bonaventura et al. 2013) and MS4 Me (Seo et al. 2013), provide state diagram editors as a visual alternative to traditional code for defining atomic models. Another strategy is to promote DEVS ideas in the layout of a simulation environment, an example being the separation of model and simulator reflected in the DesignDEVS interface (Goldstein et al. 2018).

To make the formalism more intuitive, DEVS terminology is sometimes modified. Such modifications can be as simple as replacing the term "coupled model" with "composite model", as in CoSMoS (Sarjoughian and Elamvazhuthi 2009), avoiding the possible misinterpretation that "couple" implies there only two components. Alternatively one can replace the entire vocabulary, as in OMNeT++ (Varga and Hornig 2008), which is essentially a DEVS-based tool even though it is not framed as such. Maleki et al. (2015) propose a comprehensive set of alternative terms that describe DEVS but are intended to map more closely to concepts familiar to modern-day end-user programmers. Any decision to depart from conventional

nomenclature involves a trade-off: on the one hand, inconsistencies may be introduced into the literature; on the other hand, terms observed to be potential sources of confusion can be replaced.

A final strategy for making DEVS more approachable is to emphasize various symmetries inherent in the theory. In general, it is observed that symmetric forms receive more attention than asymmetric forms and are more easily recognized and recalled (Lidwell et al. 2010). An example of symmetry can be found in the user manual of DEVS++ (Hwang 2007), where the atomic models of (1) are re-expressed as in (2).

$$\langle X, Y, S, s_0, \tau, \delta_x, \delta_y \rangle \mid \text{DEVS++ atomic model} \tag{2}$$

In (2), the Classic DEVS functions $\lambda$ and $\delta_{int}$ have been merged into a single function $\delta_y$. Combining functions does not by itself produce a simpler formalism, since the resulting function may be more complex than either of the original two. However in this case, the merging, renaming, and reordering of elements establishes a symmetry between the input/output sets $X$, $Y$ and the associated transition functions $\delta_x$, $\delta_y$. It is thus likely that a domain expert without prior exposure to DEVS would find the atomic model of (2) more approachable than that of (1). On the other hand, specifications based on (2) contrast with the literature and may be less convenient to check for certain mathematical properties. Nevertheless, if the formalism is expanded to include initialization elements—an example being the initial state $s_0$ in (2)—and many more elements associated with ABM, the benefits of symmetry become worthy of serious consideration.

### 2.2 DEVS and Dataflow Programming

At its simplest, dataflow programming implies the use of a directed graph in which every node represents a function and links direct data from the output of one node to the input of another (Davis and Keller 1982). The paradigm is often implemented in conjunction with visual programming environments that allow end-user programmers to navigate and edit dataflow graphs in 2D. The popularity of dataflow programming environments is explored by Doore et al. (2015) as an opportunity to introduce modeling and simulation into academic programs such as multimedia where discrete events, state transitions, and other foundational concepts have traditionally received little attention.

There are several ways to combine discrete event simulation and dataflow programming, a number of which are demonstrated by the Ptolemy II framework (Lee et al. 2004). In Ptolemy II, a dataflow graph can be used as a function to specify how a discrete event model responds to inputs. In this approach, neither paradigm need be substantially altered. Another option is to define a Ptolemy II discrete event model using a dataflow component with a nonzero period parameter, which imposes a fixed time step between transitions regardless of the timing of the inputs. This paradigm-mixing approach contrasts sharply with DEVS, where delays between inputs and outputs depend on the definition of the *ta* function.

Maleki et al. (2015) propose a visual interface that combines dataflow and discrete event elements into a single graph, yet separates the execution of the graph into dataflow and discrete event phases. First, an initialization phase based on dataflow programming incorporates parameters and prepares the simulation. Second, a simulation phase based on DEVS processes all timed events. Third, a finalization phase reverts to dataflow to gather information from the simulation and compute statistics. In essence, all static data is handled by dataflow programming and all time-varying data is handled by discrete event simulation.

### 2.3 DEVS and Agent-Based Modeling

Yilmaz and Ören (2009) distinguish between ***simulation for agents*** and ***agents for simulation***, only the latter of which encompasses agent-based modeling. To illustrate, consider a simulation of customers waiting in a queue. Suppose the queue is represented as a DEVS model, but the customers are tracked only via state variables and messages. This could be considered an example of "simulation for agents"—or multi-agent simulation—assuming the customers are interpreted as agents regardless of how they are modeled. The example is not, however, a case of "agents for simulation"—and hence not an example of ABM—since there is no single-customer model that drives agent behavior. But now suppose the simulation is modified

such that an additional queue is generated, by instantiating the single-queue DEVS model, whenever the existing queues are all full. For our purposes, this enhanced version is an example of ABM in which the queues are modeled as agents. From an ABM perspective, the customers are not agents at all.

Classic DEVS offers very limited support for ABM as defined above. This is partly because coupled models tend to involve a small number of explicitly named components, but more fundamentally because the number of components remains constant throughout a simulation. In order to treat the components of a coupled model as agents, dynamic structure is required. The original means of achieving dynamic structure in DEVS is to incorporate links and components into the time-varying state of an encompassing model called the network executive (Barros 1995; Zeigler et al. 2000). Subsequent developments have followed one of two strategies: distributed or centralized (Barros 2014). The distributed approach omits the executive and instead requires all changes to a network to be initiated by its components (Hu et al. 2005; Muzy and Zeigler 2014). The centralized approach retains the network executive, as in the HyFlow formalism (Barros 2016), or reformulates it and the components as macro-level and micro-level communicating instances with their own interfaces, state sets, and behavior. In the multi-level modeling formalism ML-DEVS (Steiniger et al. 2012; Steiniger and Uhrmacher 2016), an example of a centralized approach that reformulates the network executive, communication occurs through port-to-port links as well as separate upward (micro-to-macro) and downward (macro-to-micro) interaction mechanisms.

Regardless of which variant is used, introducing dynamic structure into DEVS coupled models entails a large and possibly intimidating set of mathematical elements. Are there simpler ways to make DEVS amenable to ABM? Exploring this question, we first observe that most programming languages provide compositions and collections as separate data types (e.g. a C++ class vs. an array), and that the run-time creation and termination of agents would most easily be achieved using a collection data type. The analogous approach in DEVS would be to leave the composite (coupled) model as a static entity, and introduce a separate collection model containing a variable number of instances of an agent model. Managing links to newly created agents is known to be a difficult challenge (Steiniger and Uhrmacher 2016), yet one can dispense with links and transfer messages to/from agents by associating them with agent IDs. The invention closest to this idea is Vectorial DEVS (Bergero and Kofman 2014), which includes arrays containing an arbitrary but fixed number of DEVS model instances. A more versatile collection model would be needed to permit the mid-simulation creation and termination of agents and thus improve support for ABM.

## 3 OVERVIEW OF SYMMETRIC DEVS

Symmetric DEVS is based on Classic DEVS, but features a more symmetric set of mathematical elements accommodating not only discrete event simulation but also dataflow programming and ABM. Instead of atomic and coupled ***models***, Symmetric DEVS specifications consist of atomic, function, composite and collection ***nodes***. The main elements of all four types of nodes are illustrated in Figure 1.

It is in the atomic nodes that the symmetry which characterizes the formalism is most noticeable. Ports are classified as either ***flow*** or ***message*** ports, and also as ***input*** or ***output*** ports. Combining these attributes yields four types of ports. When representing nodes visually, as in Figure 1, we always position each type of port on a particular side of the node: left/right for flow input/output ports; top/bottom for message input/output ports. Each type of port is associated with its own event handler: initialization/finalization for flow input/output ports; unplanned/planned for message input/output ports. All event handlers except initialization depend on the ***elapsed duration*** measured since the preceding event. All event handlers except finalization supply a ***planned duration*** measured until the next scheduled planned event. Function nodes are also highly symmetric, but feature only flow input/output ports and a single event handler for dataflow events. The flow/dataflow elements involve neither state transitions nor the advancement of simulated time, and are hence unrelated to the flow values of the HyFlow formalism where state is omnipresent.

Composite and collection nodes share the same external interface as atomic nodes, with the four types of ports, but differ internally. They both support hierarchical model design by encapsulating components or agents described by node specifications of any type.

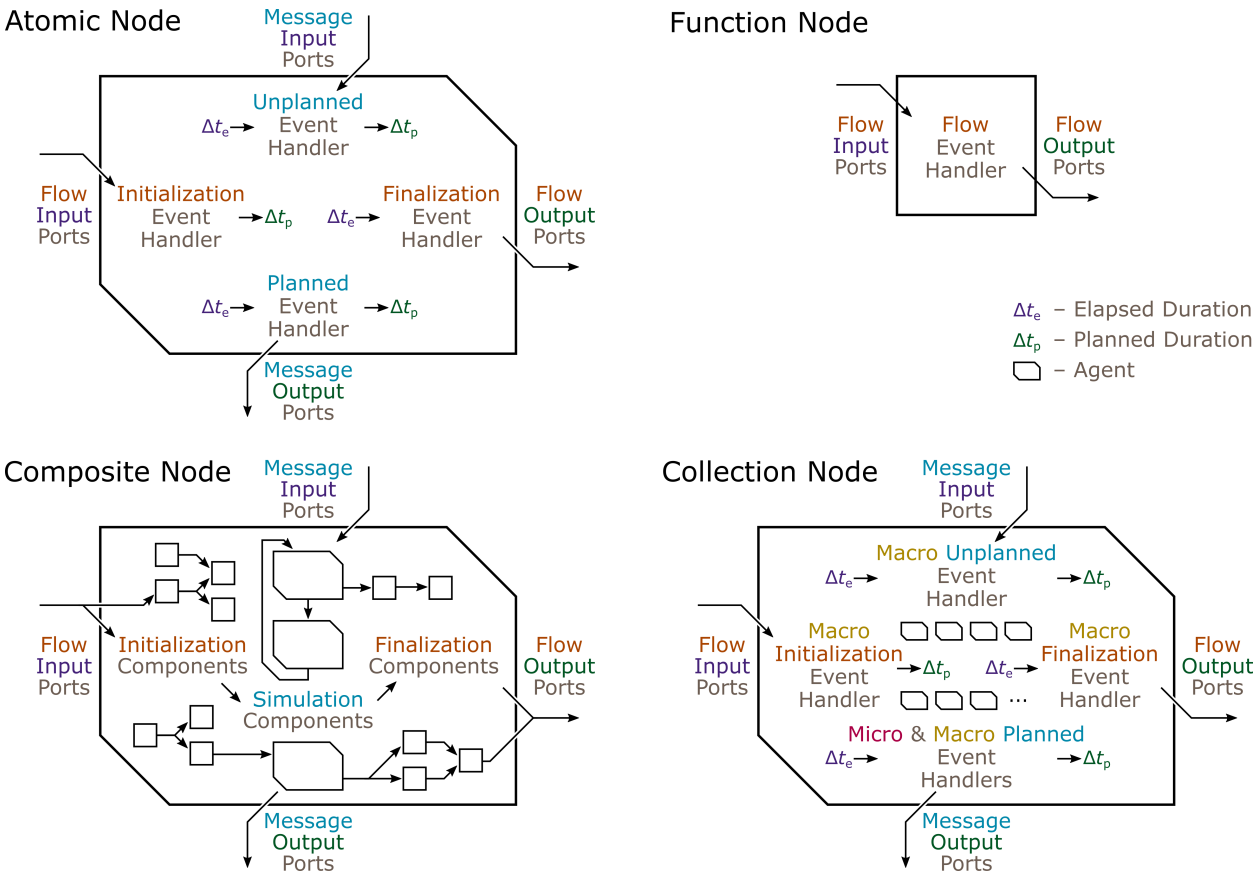


Figure 1: The four types of Symmetric DEVS nodes: atomic, function, composite, and collection.

The composite nodes are based on the work of Maleki et al. (2015) and feature two overlapping networks. The dataflow network is a horizontally oriented directed acyclic graph that guides information through the composite node's flow ports and the flow ports of all component nodes. The simulation network is a vertically oriented directed graph (cycles permitted) that guides information through the composite node's message ports and the message ports of a subset of the component nodes (the "simulation components" in Figure 1). Depending on the design of the dataflow network, the flow input/output ports of the simulation components may have a one-to-one correspondence with those of the surrounding composite node, or they may be completely different. By separating the inner and outer interfaces in this fashion, the dataflow network allows the structure of the composite node to be fully encapsulated.

The collection node borrows ideas from a number of sources. As in Vectorial DEVS (Bergero and Kofman 2014), agents are nodes that share a single specification but may be assigned different parameter (flow input) values. All agent interactions are achieved via the juxtaposition of agent IDs with the information to be communicated. Unlike Vectorial DEVS, but similar to ML-DEVS (Steiniger and Uhrmacher 2016), the micro-level behavior of the agent nodes is complemented with macro-level behavior associated with the collection as a whole. The macro-level behavior is described by event handlers similar to those of the atomic node, but (*a*) renamed with the prefix ***macro***, (*b*) extended with a ***micro*** planned event handler, and (*c*) enhanced to specify the creation of, interaction with, and termination of agents. The upper-level event handlers allow the agent IDs, and hence the structure of the collection node, to be encapsulated.

While the diagrams in Figure 1 depict the most prominent elements of the four types of Symmetric DEVS nodes, Section 4 presents the complete set of elements along with their proposed notations and an informal description of their semantics.

## 4 FORMAL DEFINITION OF SYMMETRIC DEVS NODES

### 4.1 Common Definitions

A number of mathematical elements are common to most if not all types of Symmetric DEVS nodes. These common elements include the functions below, which define a node's ports by mapping each port's identifier (ID) to the set of values that can be communicated through the port.

| $P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}$ | port definition functions |
|---|---|
| $P_{\text{fi}}(pid_{\text{fi}}) = \mathbb{X}_{\text{fi}}$ | $pid_{\text{fi}}$ is a flow input port ID; $\mathbb{X}_{\text{fi}}$ is the set of values for port $pid_{\text{fi}}$ |
| $P_{\text{mi}}(pid_{\text{mi}}) = \mathbb{X}_{\text{mi}}$ | $pid_{\text{mi}}$ is a message input port ID; $\mathbb{X}_{\text{mi}}$ is the set of values for port $pid_{\text{mi}}$ |
| $P_{\text{mo}}(pid_{\text{mo}}) = \mathbb{X}_{\text{mo}}$ | $pid_{\text{mo}}$ is a message output port ID; $\mathbb{X}_{\text{mo}}$ is the set of values for port $pid_{\text{mo}}$ |
| $P_{\text{fo}}(pid_{\text{fo}}) = \mathbb{X}_{\text{fo}}$ | $pid_{\text{fo}}$ is a flow output port ID; $\mathbb{X}_{\text{fo}}$ is the set of values for port $pid_{\text{fo}}$ |

There are also elements that exist only during a simulation. These include the states and time durations associated with intervals between events. As indicated below, state and duration elements are often grouped together. There are also values that become available on individual message ports. Such values are often grouped with the port ID.

| | |
|---|---|
| $[s, \Delta t_{\text{e}}]$ | $s$ is the state leading up to an event; $\Delta t_{\text{e}}$ is the elapsed duration since the previous event |
| $[s', \Delta t_{\text{p}}]$ | $s'$ is the state following an event; $\Delta t_{\text{p}}$ is the planned duration until the next planned event |
| $[pid_{\text{mi}}, x_{\text{mi}}]$ | $x_{\text{mi}}$ is a value on message input port $pid_{\text{mi}}$; $x_{\text{mi}} \in P_{\text{mi}}(pid_{\text{mi}})$ |
| $[pid_{\text{mo}}, x_{\text{mo}}]$ | $x_{\text{mo}}$ is a value on message output port $pid_{\text{mo}}$; $x_{\text{mo}} \in P_{\text{mo}}(pid_{\text{mo}})$ |

Unlike messages, which become available on ***individual*** message ports, flow values tend to become available on ***all*** flow input or flow output ports. Functions are therefore used to map flow port IDs to values.

| $X_{\text{fi}}, X_{\text{fo}}$ | port value functions |
|---|---|
| $X_{\text{fi}}(pid_{\text{fi}}) = x_{\text{fi}}$ | $x_{\text{fi}}$ is a value on flow input port $pid_{\text{fi}}$; $x_{\text{fi}} \in P_{\text{fi}}(pid_{\text{fi}})$ |
| $X_{\text{fo}}(pid_{\text{fo}}) = x_{\text{fo}}$ | $x_{\text{fo}}$ is a value on flow output port $pid_{\text{fo}}$; $x_{\text{fo}} \in P_{\text{fo}}(pid_{\text{fo}})$ |

The expression $\mathcal{D}(f)$ represents the domain of function $f$. For example, $\mathcal{D}(P_{\text{fi}})$ is a mathematical set containing a node's flow input port IDs. Port and component IDs are symbols enclosed in double quotes.

### 4.2 Atomic Node Definition

Atomic node specifications consist of four port definition functions, as in Section 4.1, plus the four event handling functions elaborated below. The eight elements exhibit a symmetry mirroring that of the corresponding diagram in Figure 1.

| $\Psi = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, f_{\text{init}}, f_{\text{u}}, f_{\text{p}}, f_{\text{final}} \rangle$ | atomic node definition |
|---|---|
| $f_{\text{init}}(X_{\text{fi}}) = [s', \Delta t_{\text{p}}]$ | initialization function |
| $f_{\text{u}}([s, \Delta t_{\text{e}}], [pid_{\text{mi}}, x_{\text{mi}}]) = [s', \Delta t_{\text{p}}]$ | unplanned event function |
| $f_{\text{p}}([s, \Delta t_{\text{e}}]) = [[s', \Delta t_{\text{p}}], [pid_{\text{mo}}, x_{\text{mo}}]]$ | planned event function |
| $f_{\text{final}}([s, \Delta t_{\text{e}}]) = X_{\text{fo}}$ | finalization function |

The simulation of an atomic node begins when the initialization function $f_{\text{init}}$ takes the flow input (parameter) values in $X_{\text{fi}}$ and produces the initial state and the planned duration $\Delta t_{\text{p}}$. If a message input value $x_{\text{mi}}$ is received on any message input port $pid_{\text{mi}}$ either when $\Delta t_{\text{p}}$ elapses or at an earlier time, the unplanned event function $f_{\text{u}}$ is invoked to update the state and supply a new planned duration. If $\Delta t_{\text{p}}$ elapses before an input is received, the planned event function $f_{\text{p}}$ is invoked to update the state and planned duration, and possibly to send a message output value $x_{\text{mo}}$ on message output port $pid_{\text{mo}}$. If $pid_{\text{mo}} = \varnothing$, no message is sent. At the end of a simulation, or if the atomic node is an agent being terminated, the finalization function $f_{\text{final}}$ is invoked to produce the flow output (statistic) values in $X_{\text{fo}}$.

### 4.3 Function Node Definition

Function node specifications consist of the two port definition functions associated with dataflow programming, plus a single event handling function.

$$\Psi = \langle P_{\text{fi}}, P_{\text{fo}}, f \rangle \mid \text{function node definition}$$
$$f(X_{\text{fi}}) = X_{\text{fo}} \mid \text{flow event function}$$

When the flow input values in $X_{\text{fi}}$ become available, $f$ may be invoked to produce the flow output values in $X_{\text{fo}}$. Invocation occurs during the initialization or finalization phase of an encompassing composite node. It is possible to specify pure dataflow networks using composite or collection nodes with no atomic nodes at any depth in the hierarchy. Such composite/collection nodes essentially act as function nodes.

### 4.4 Composite Node Definition

Both composite and collection nodes support hierarchical specifications by encapsulating other nodes of any type. In this context the symbol M indicates macro-level elements associated with the overall composition or collection, whereas the symbol $\mu$ indicates micro-level elements associated with the component or agent nodes. The component function $\psi_\mu$, which includes a subscript $\mu$ to associate it with the lower level, maps the ID of each of a composite node's components to the specification of the component.

$$\psi_\mu \mid \text{component function}$$
$$\psi_\mu(cid) = \Psi_\mu \mid cid \text{ is a component ID; } \Psi_\mu \text{ is the definition of component } cid$$

Composite node specifications consist of all four port definition functions, the component function described above, and six sets of links. The link elements include sets of inward (macro-to-micro), inner (micro-to-micro), and outward (micro-to-macro) links for both the dataflow network ($\mathbb{L}_{\text{fi}}$, $\mathbb{L}_{\text{f}\mu}$, $\mathbb{L}_{\text{fo}}$) and the simulation network ($\mathbb{L}_{\text{mi}}$, $\mathbb{L}_{\text{m}\mu}$, $\mathbb{L}_{\text{mo}}$).

$$\Psi_{\text{M}} = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, \psi_\mu, \mathbb{L}_{\text{fi}}, \mathbb{L}_{\text{f}\mu}, \mathbb{L}_{\text{fo}}, \mathbb{L}_{\text{mi}}, \mathbb{L}_{\text{m}\mu}, \mathbb{L}_{\text{mo}} \rangle \mid \text{composite node definition}$$

| | |
|---|---|
| $\mathbb{L}_{\text{fi}} = \{\ldots, [pid_{\text{fi}}, [cid, pid_{\text{fi}}]], \ldots\}$ | set of inward flow links |
| $\mathbb{L}_{\text{f}\mu} = \{\ldots, [[cid, pid_{\text{fo}}], [cid, pid_{\text{fi}}]], \ldots\}$ | set of inner flow links |
| $\mathbb{L}_{\text{fo}} = \{\ldots, [[cid, pid_{\text{fo}}], pid_{\text{fo}}], \ldots\}$ | set of outward flow links |
| $\mathbb{L}_{\text{mi}} = \{\ldots, [pid_{\text{mi}}, [cid, pid_{\text{mi}}]], \ldots\}$ | set of inward message links |
| $\mathbb{L}_{\text{m}\mu} = \{\ldots, [[cid, pid_{\text{mo}}], [cid, pid_{\text{mi}}]], \ldots\}$ | set of inner message links |
| $\mathbb{L}_{\text{mo}} = \{\ldots, [[cid, pid_{\text{mo}}], pid_{\text{mo}}], \ldots\}$ | set of outward message links |

Composite nodes feature three distinct types of components. An initialization component (*a*) is a function node or a composite/collection node that contains no simulation components/agents, and (*b*) is not downstream of any simulation component. A simulation component is an atomic node or a composite/collection node that contains at least one simulation component/agent. A finalization component (*a*) is a function node or a composite/collection node that contains no simulation components/agents, (*b*) is downstream of at least one simulation component, and (*c*) is not upstream of any simulation component.

Composite nodes are executed in three phases. In the initialization phase, the initialization components are executed whenever they receive all flow input values through their incoming flow links. The simulation components are then initialized. In the simulation phase, the simulation components' planned and unplanned event functions are invoked as simulated time advances. If a message input value is received, it is propagated via the inward message links to the components, triggering unplanned events. Otherwise, if a planned duration elapses for any component, it undergoes a planned event. Any message produced by that component gets propagated along the inner message links to other components, triggering unplanned events. The message may also be directed out of the composite node via an outward message link. In the finalization phase, the simulation components are finalized, then the finalization components are executed whenever they receive all flow input values through their incoming flow links.

### 4.5 Collection Node Definition

Whereas composite nodes contain a fixed number of components but permit an assortment of component specifications, collection nodes contain an arbitrary and time-varying number of agents but permit only a single agent specification. Collection nodes introduce a number of elements, including the following.

| | |
|---|---|
| $aid$ | an agent ID |
| $\Psi_\mu$ | the definition of all agents in the collection node |
| $\mathbb{F}$ | a set of IDs of agents to be terminated and removed from the collection node |

Agents in a collection node cannot exchange data directly with one another. Instead, all communication occurs between the macro (collection) and micro (agent) levels using the four elements below.

| $\mu_{\text{fi}}, \mu_{\text{mi}}, \mu_{\text{mo}}, \mu_{\text{fo}}$ | micro port value elements |
|---|---|
| $\mu_{\text{fi}}(aid) = X_{\text{fi}}^{\mu}$ | $X_{\text{fi}}^{\mu}$ is the flow input port value function of new agent $aid$ |
| $\mu_{\text{mi}}(aid) = [pid_{\text{mi}}^{\mu}, x_{\text{mi}}^{\mu}]$ | $x_{\text{mi}}^{\mu}$ is a value on message input port $pid_{\text{mi}}^{\mu}$ of existing agent $aid$ |
| $\mu_{\text{mo}} = [aid, [pid_{\text{mo}}^{\mu}, x_{\text{mo}}^{\mu}]]$ | $x_{\text{mo}}^{\mu}$ is a value on message output port $pid_{\text{mo}}^{\mu}$ of existing agent $aid$ |
| $\mu_{\text{fo}}(aid) = X_{\text{fo}}^{\mu}$ | $X_{\text{fo}}^{\mu}$ is the flow output port value function of existing agent $aid$ |

Collection node specifications consist of all four port definition functions, the agent specification, and five event handling functions.

| $\Psi_{\text{M}} = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, \Psi_\mu, f_{\text{init}}^{\text{M}}, f_{\text{u}}^{\text{M}}, f_{\text{p}}^{\mu}, f_{\text{p}}^{\text{M}}, f_{\text{final}}^{\text{M}} \rangle$ | collection node definition |
|---|---|
| $f_{\text{init}}^{\text{M}}(X_{\text{fi}}) = [[s', \Delta t_{\text{p}}], [\mu_{\text{fi}}, \mu_{\text{mi}}]]$ | macro initialization function |
| $f_{\text{u}}^{\text{M}}([s, \Delta t_{\text{e}}], [pid_{\text{mi}}, x_{\text{mi}}], \mu_{\text{fo}}) = [[s', \Delta t_{\text{p}}], \mathbb{F}, [\mu_{\text{fi}}, \mu_{\text{mi}}]]$ | macro unplanned event function |
| $f_{\text{p}}^{\mu}([s, \Delta t_{\text{e}}], \mu_{\text{mo}}, \mu_{\text{fo}}) = [[s', \Delta t_{\text{p}}], [pid_{\text{mo}}, x_{\text{mo}}], \mathbb{F}, [\mu_{\text{fi}}, \mu_{\text{mi}}]]$ | micro planned event function |
| $f_{\text{p}}^{\text{M}}([s, \Delta t_{\text{e}}], \mu_{\text{fo}}) = [[s', \Delta t_{\text{p}}], [pid_{\text{mo}}, x_{\text{mo}}], \mathbb{F}, [\mu_{\text{fi}}, \mu_{\text{mi}}]]$ | macro planned event function |
| $f_{\text{final}}^{\text{M}}([s, \Delta t_{\text{e}}], \mu_{\text{fo}}) = X_{\text{fo}}$ | macro finalization function |

The simulation of a collection node begins when the macro initialization function takes the flow input values in $X_{\text{fi}}$ and produces the initial state and $\Delta t_{\text{p}}$. This function can also create and send messages to agents, in that order, using $\mu_{\text{fi}}$ and $\mu_{\text{mi}}$. If a message input value $x_{\text{mi}}$ is received on any message input port $pid_{\text{mi}}$ either when $\Delta t_{\text{p}}$ elapses or at an earlier time, the macro unplanned event function is invoked to update the state and supply a new $\Delta t_{\text{p}}$. The macro unplanned event function can also terminate, create, and send messages to agents, in that order, using $\mathbb{F}$, $\mu_{\text{fi}}$, and $\mu_{\text{mi}}$. If $\Delta t_{\text{p}}$ elapses before an input is received, the macro planned event function is invoked to update the state and $\Delta t_{\text{p}}$, and possibly to send a message output value $x_{\text{mo}}$ on message output port $pid_{\text{mo}}$ (unless $pid_{\text{mo}} = \varnothing$, in which case no message is sent). The macro planned event function can also terminate, create, and send messages to agents. At the end of a simulation, or if the collection node is itself an agent being terminated, the macro finalization function is invoked to produce the flow output values in $X_{\text{fo}}$.

There is also a micro planned event function, invoked after an agent sends a message during its planned event. The message is made available via $\mu_{\text{mo}}$. Apart from how it is triggered, a micro planned event is similar to a macro planned event. It updates the state and supplies a new planned duration, it may send a message out of the collection, and it may terminate, create, and send messages to agents. If tied, macro unplanned events occur before micro planned events, which occur before macro planned events.

When an agent is terminated, its flow output values must be available to the collection node. Yet $\mu_{\text{fo}}$ provides flow output values for every agent that ***might*** get terminated during an event. Implementations of Symmetric DEVS can make use of two-way communication between macro- and micro-level events; an agent can provide its flow output values ***only*** if its termination is signaled. But in a mathematical context, dependent events are handled strictly in sequence, so flow output values must always be available for all preexisting agents. Because agents in $\mathbb{F}$ are terminated before agents in $\mu_{\text{fi}}$ are created, it is impossible to create and terminate an agent in a single event and thereby render its flow output values inaccessible.

## 5 EXAMPLES

Here we provide a simple example of each of the four types of nodes.

For the ***atomic node***, we specify a queueing_node which receives job IDs on the $jid_{\text{in}}$ message input port, queues them, processes the first job for a fixed service duration, then outputs the ID of the completed job on the $jid_{\text{out}}$ message output port. The node accepts the service duration $\Delta t_{\text{serv}}$ as a flow input. It produces the idle duration $\Delta t_{\text{idle}}$, the total time that the queue is empty with no job being processed, as a flow output. The complete formal specification is below.

$$\frac{\Psi_{\text{queueing\_node}} = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, f_{\text{init}}, f_{\text{u}}, f_{\text{p}}, f_{\text{final}} \rangle}{P_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}) = P_{\text{fo}}(\text{“}\Delta t_{\text{idle}}\text{”}) = \mathbb{R}_0^+; \quad P_{\text{mi}}(\text{“}jid_{\text{in}}\text{”}) = P_{\text{mo}}(\text{“}jid_{\text{out}}\text{”}) = \mathbb{N}}$$

$$\frac{f_{\text{init}}(X_{\text{fi}}) = [[\Delta t_{\text{serv}}, Q, \Delta t_{\text{idle}}, \Delta t_{\text{p}}], \Delta t_{\text{p}}]}{\Delta t_{\text{serv}} = X_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}); \quad Q = [\,]; \quad \Delta t_{\text{idle}} = 0; \quad \Delta t_{\text{p}} = \infty}$$

$$\frac{f_{\text{u}}([[\Delta t_{\text{serv}}, Q, \Delta t_{\text{idle}}, \Delta t_{\text{p}}], \Delta t_{\text{e}}], [\text{“}jid_{\text{in}}\text{”}, jid]) = [[\Delta t_{\text{serv}}, Q', \Delta t_{\text{idle}}', \Delta t_{\text{p}}'], \Delta t_{\text{p}}']}{Q' = Q \mathbin{||} [jid]; \quad \Delta t_{\text{idle}}' = \Delta t_{\text{idle}} + \begin{cases} \Delta t_{\text{e}} & \text{if } \#Q = 0 \\ 0 & \text{if } \#Q > 0 \end{cases}; \quad \Delta t_{\text{p}}' = \begin{cases} \Delta t_{\text{serv}} & \text{if } \#Q' = 1 \\ \Delta t_{\text{p}} - \Delta t_{\text{e}} & \text{if } \#Q' > 1 \end{cases}}$$

$$\frac{f_{\text{p}}([[\Delta t_{\text{serv}}, Q, \Delta t_{\text{idle}}, \Delta t_{\text{p}}], \Delta t_{\text{e}}]) = [[[\Delta t_{\text{serv}}, Q', \Delta t_{\text{idle}}, \Delta t_{\text{p}}'], \Delta t_{\text{p}}'], [\text{“}jid_{\text{out}}\text{”}, jid]]}{Q' = Q : (1..\#Q); \quad \Delta t_{\text{p}}' = \begin{cases} \infty & \text{if } \#Q' = 0 \\ \Delta t_{\text{serv}} & \text{if } \#Q' > 0 \end{cases}}$$

$$\frac{f_{\text{final}}([[\Delta t_{\text{serv}}, Q, \Delta t_{\text{idle}}, \Delta t_{\text{p}}], \Delta t_{\text{e}}]) = X_{\text{fo}}}{X_{\text{fo}}(\text{“}\Delta t_{\text{idle}}\text{”}) = \Delta t_{\text{idle}} + \begin{cases} \Delta t_{\text{e}} & \text{if } \#Q = 0 \\ 0 & \text{if } \#Q > 0 \end{cases}}$$

The state variable $Q$ is the vector of job IDs in the queue. It is initially empty ($Q = [\,]$), grows as new job IDs ($jid$) are received ($Q' = Q \mathbin{||} [jid]$ appends an ID; the notation $A \mathbin{||} B$ represents the concatenation of vectors $A$ and $B$), and shrinks as jobs are completed ($Q' = Q : (1..\#Q)$ removes the ID at index 0; the notation $A : (b..c)$ represents a slice of vector $A$ corresponding to the range of zero-based indices $i$ for which $b \le i < c$; the notation $\#A$ represents the length of vector $A$).

For the ***function node***, we specify a plus_node which simply adds the real numbers it receives on flow input ports $a$ and $b$ and places the sum on flow output port $c$.

$$\frac{\Psi_{\text{plus\_node}} = \langle P_{\text{fi}}, P_{\text{fo}}, f \rangle}{P_{\text{fi}}(\text{“}a\text{”}) = P_{\text{fi}}(\text{“}b\text{”}) = P_{\text{fo}}(\text{“}c\text{”}) = \mathbb{R}}$$

$$\frac{f(X_{\text{fi}}) = X_{\text{fo}}}{a = X_{\text{fi}}(\text{“}a\text{”}); \quad b = X_{\text{fi}}(\text{“}b\text{”}); \quad c = a + b; \quad X_{\text{fo}}(\text{“}c\text{”}) = c}$$

Our ***composite node*** example is a two_stage_queueing_node consisting of two queueing_node components connected in series. The interface is the same as the queueing_node, but the behavior differs as a job must pass through both internal queues ($\text{queue}_{\text{A}}$ and $\text{queue}_{\text{B}}$) before it is fully processed. The service duration parameter $\Delta t_{\text{serv}}$ is directed via inward flow links to both internal queues. However the idle duration statistics $\Delta t_{\text{idle}}$ produced by $\text{queue}_{\text{A}}$ and $\text{queue}_{\text{B}}$ are propagated via inner flow links to a plus_node finalization component, which adds them together. The total idle duration is then directed out of the composite node via an outward flow link. The specification thus features both a simulation network

transporting job ID messages, and a dataflow network handling static parameters and statistics.

$$\frac{\Psi_{\text{two\_stage\_queueing\_node}} = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, \psi_\mu, \mathbb{L}_{\text{fi}}, \mathbb{L}_{\text{f}\mu}, \mathbb{L}_{\text{fo}}, \mathbb{L}_{\text{mi}}, \mathbb{L}_{\text{m}\mu}, \mathbb{L}_{\text{mo}} \rangle}{\begin{array}{l} P_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}) = P_{\text{fo}}(\text{“}\Delta t_{\text{idle}}\text{”}) = \mathbb{R}_0^+; \quad P_{\text{mi}}(\text{“}jid_{\text{in}}\text{”}) = P_{\text{mo}}(\text{“}jid_{\text{out}}\text{”}) = \mathbb{N} \\ \psi_\mu(\text{“queue}_{\text{A}}\text{”}) = \psi_\mu(\text{“queue}_{\text{B}}\text{”}) = \Psi_{\text{queueing\_node}}; \quad \psi_\mu(\text{“plus”}) = \Psi_{\text{plus\_node}} \end{array}}$$

$$\begin{array}{l} \mathbb{L}_{\text{fi}} = \{[\text{“}\Delta t_{\text{serv}}\text{”}, [\text{“queue}_{\text{A}}\text{”}, \text{“}\Delta t_{\text{serv}}\text{”}]], [\text{“}\Delta t_{\text{serv}}\text{”}, [\text{“queue}_{\text{B}}\text{”}, \text{“}\Delta t_{\text{serv}}\text{”}]]\} \\ \mathbb{L}_{\text{f}\mu} = \{[[\text{“queue}_{\text{A}}\text{”}, \text{“}\Delta t_{\text{idle}}\text{”}], [\text{“plus”}, \text{“}a\text{”}]], [[\text{“queue}_{\text{B}}\text{”}, \text{“}\Delta t_{\text{idle}}\text{”}], [\text{“plus”}, \text{“}b\text{”}]]\} \\ \mathbb{L}_{\text{fo}} = \{[[\text{“plus”}, \text{“}c\text{”}], \text{“}\Delta t_{\text{idle}}\text{”}]\} \\ \mathbb{L}_{\text{mi}} = \{[\text{“}jid_{\text{in}}\text{”}, [\text{“queue}_{\text{A}}\text{”}, \text{“}jid_{\text{in}}\text{”}]]\} \\ \mathbb{L}_{\text{m}\mu} = \{[[\text{“queue}_{\text{A}}\text{”}, \text{“}jid_{\text{out}}\text{”}], [\text{“queue}_{\text{B}}\text{”}, \text{“}jid_{\text{in}}\text{”}]]\} \\ \mathbb{L}_{\text{mo}} = \{[[\text{“queue}_{\text{B}}\text{”}, \text{“}jid_{\text{out}}\text{”}], \text{“}jid_{\text{out}}\text{”}]\} \end{array}$$

Our ***collection node*** example is a parallel_queueing_node that at first contains one queue, but creates an additional parallel queue whenever a job is received while the existing queues are all full. The internal queues, defined as queueing_node agents, are considered full if they contain $n_{\max}$ jobs, where $n_{\max}$ is a parameter. The specification makes use of $\mu_{\text{fi}}$ to create queues, $\mu_{\text{mi}}$ and $\mu_{\text{mo}}$ to exchange messages with them, and $\mu_{\text{fo}}$ to obtain idle durations. The macro-level state variable $N$ keeps track of the number of jobs in each queue. It is intended that low-fidelity whole-system representations such as $N$ be employed at the macro level, duplicating some but not all of the high-fidelity micro-level information. Note that $find_{\text{queue}}$ yields either (*a*) the index of the first queue with space for another job, or (*b*) the index of a new queue.

$$\frac{\Psi_{\text{parallel\_queueing\_node}} = \langle P_{\text{fi}}, P_{\text{mi}}, P_{\text{mo}}, P_{\text{fo}}, \Psi_\mu, f^{\text{M}}_{\text{init}}, f^{\text{M}}_{\text{u}}, f^{\mu}_{\text{p}}, f^{\text{M}}_{\text{p}}, f^{\text{M}}_{\text{final}} \rangle}{\begin{array}{l} P_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}) = P_{\text{fo}}(\text{“}\Delta t_{\text{idle}}\text{”}) = \mathbb{R}_0^+; \quad P_{\text{fi}}(\text{“}n_{\max}\text{”}) = \mathbb{N}^+; \quad P_{\text{mi}}(\text{“}jid_{\text{in}}\text{”}) = P_{\text{mo}}(\text{“}jid_{\text{out}}\text{”}) = \mathbb{N} \\ \Psi_\mu = \Psi_{\text{queueing\_node}} \end{array}}$$

$$\frac{f^{\text{M}}_{\text{init}}(X_{\text{fi}}) = [[[\Delta t_{\text{serv}}, n_{\max}, N], \infty], [\mu_{\text{fi}}, \mu_{\text{mi}}]]}{\Delta t_{\text{serv}} = X_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}); \quad n_{\max} = X_{\text{fi}}(\text{“}n_{\max}\text{”}); \quad N = [0]; \quad \mu_{\text{fi}}(0) = X^{\mu}_{\text{fi}}; \quad X^{\mu}_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}) = \Delta t_{\text{serv}}}$$

$$\frac{f^{\text{M}}_{\text{u}}([[[\Delta t_{\text{serv}}, n_{\max}, N], \Delta t_{\text{e}}], [\text{“}jid_{\text{in}}\text{”}, jid], \mu_{\text{fo}}) = [[[\Delta t_{\text{serv}}, n_{\max}, N'], \infty], \varnothing, [\mu_{\text{fi}}, \mu_{\text{mi}}]]}{\begin{array}{l} aid = find_{\text{queue}}(N \mathbin{||} [0], 0); \quad find_{\text{queue}}(\hat{N}, i) = \begin{cases} i & \text{if } \hat{N}(i) < n_{\max} \\ find_{\text{queue}}(\hat{N}, i+1) & \text{if } \hat{N}(i) = n_{\max} \end{cases} \\ \forall_{i \in (\mathcal{D}(N) - \{aid\})}(N'(i) = N(i)); \quad N'(aid) = \begin{cases} N(aid) + 1 & \text{if } aid < \#N \\ 1 & \text{if } aid = \#N \end{cases} \\ aid = \#N \implies \mu_{\text{fi}}(aid) = X^{\mu}_{\text{fi}}; \quad X^{\mu}_{\text{fi}}(\text{“}\Delta t_{\text{serv}}\text{”}) = \Delta t_{\text{serv}} \\ \mu_{\text{mi}}(aid) = [\text{“}jid_{\text{in}}\text{”}, jid] \end{array}}$$

$$\frac{f^{\mu}_{\text{p}}([[\Delta t_{\text{serv}}, n_{\max}, N], \Delta t_{\text{e}}], \mu_{\text{mo}}, \mu_{\text{fo}}) = [[[\Delta t_{\text{serv}}, n_{\max}, N'], \infty], [\text{“}jid_{\text{out}}\text{”}, jid], \varnothing, [\mu_{\text{fi}}, \mu_{\text{mi}}]]}{\begin{array}{l} [aid, [\text{“}jid_{\text{out}}\text{”}, jid]] = \mu_{\text{mo}} \\ \forall_{i \in (\mathcal{D}(N) - \{aid\})}(N'(i) = N(i)); \quad N'(aid) = N(aid) - 1 \end{array}}$$

$$\frac{f^{\text{M}}_{\text{final}}([[\Delta t_{\text{serv}}, n_{\max}, N], \Delta t_{\text{e}}], \mu_{\text{fo}}) = X_{\text{fo}}}{X_{\text{fo}}(\text{“}\Delta t_{\text{idle}}\text{”}) = \textstyle\sum_{i \in \mathcal{D}(N)}(\mu_{\text{fo}}(i))(\text{“}\Delta t_{\text{idle}}\text{”})}$$

Note: $\mathcal{D}(N)$ is the domain of $N$

Since all events in this example are triggered either by an external message or by an agent, the planned durations are always $\infty$. It follows that the macro planned event can never be invoked, and so it is left unspecified. Another simplification is that the internal queues, once created in response to a high volume of incoming jobs, are never terminated—not even if they become underutilized. As a result, $\mathbb{F}$ is always the empty set $\varnothing$, and there is no need to query $\mu_{\text{fo}}$ until the macro finalization event occurs.

## 6 DISCUSSION AND FUTURE WORK

The atomic, function, composite, and collection node examples in Section 5 have been coded, tested, and packaged with the SyDEVS library (https://autodesk.github.io/sydevs), an open source implementation of Symmetric DEVS in C++. As is typical of such a framework, SyDEVS deviates from the underlying theory in a few places. For instance, SyDEVS allows collection node event handlers to create, send messages to, and terminate agents in any order; in the formalism, such events occur strictly in sequence (e.g. agent sends message → micro planned event → agents terminated → agents created → agents receive messages). Users can choose to make use of the implementation's flexibility or adhere strictly to the formalism.

In paring down node specifications to the essential elements, the Classic DEVS state set $S$ and tie-breaking function *Select* have been omitted. For Symmetric DEVS atomic and collection nodes, one may derive $S$ as the set of all states that can result from any of the event handling functions. One can also add $S$ to these nodes on a per-application basis, but we prefer not to make this a requirement. Similarly, one may add a *Select* function to composite and collection nodes, but we propose a non-deterministic interpretation of simultaneous event ordering. If multiple components or agents are scheduled to undergo a planned event at the same time, and if no tie-breaking rule is given, then any ordering of the events should be considered consistent with the specification. Dijkstra (1975) showed that programs with non-deterministic conditional and repetition structures can yield elegant proofs. It is worth exploring whether non-deterministic tie-breaking can lead to shorter or more robust simulation model specifications.

To conclude, the Symmetric DEVS formalism extends Classic DEVS with collections, which support agent-based modeling, and dataflow elements, which initialize and finalize agents and other DEVS model instances. The approach offers a middle ground between static and dynamic structure variants of DEVS. To control conceptual complexity—a key challenge in promoting the adoption of scalable modeling practices by domain experts—we propose specifications involving minimal sets of mathematical elements exhibiting high degrees of symmetry. Future work includes formal simulation semantics and closure under coupling mappings that express composite and collection nodes in the same form as atomic or function nodes.

## REFERENCES


Barros, F. J. 1995. "Dynamic Structure Discrete Event System Specification: A New Formalism for Dynamic Structure Modeling and Simulation". In *Proceedings of the 1995 Winter Simulation Conference*, edited by C. Alexopoulos et al., 781–785. Piscataway, New Jersey: IEEE.

Barros, F. J. 2014. "On the Representation of Dynamic Topologies: The Case for Centralized and Modular Approaches". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '14, 8 pages. Tampa, Florida.

Barros, F. J. 2016. "On the Representation of Time in Modeling & Simulation". In *Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. K. Roeder et al., 1571–1582. Piscataway, New Jersey: IEEE.

Bergero, F., and E. Kofman. 2014. "A Vectorial DEVS Extension for Large Scale System Modeling and Parallel Simulation". *Simulation* 90(5):522–546.

Bonaventura, M., G. A. Wainer, and R. Castro. 2013. "Graphical Modeling and Simulation of Discrete-Event Systems with CD++Builder". *Simulation* 89(1):4–27.

Davis, A. L., and R. M. Keller. 1982. "Data Flow Program Graphs". *Computer* 15(2):26–41.

Dijkstra, E. W. 1975. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *Communications of the ACM* 18(8):453–457.

Doore, K., D. Vega, and P. Fishwick. 2015. "A Media-Rich Curriculum for Modeling and Simulation". In *Proceedings of the Principles of Advanced Discrete Simulation Conference*, SIGSIM-PADS '15, 23–34. London, UK.

Goldstein, R., S. Breslav, and A. Khan. 2018. "Practical Aspects of the DesignDEVS Simulation Environment". *Simulation* 94(4):301–328.

Hu, X., B. P. Zeigler, and S. Mittal. 2005. "Variable Structure in DEVS Component-Based Modeling and Simulation". *Simulation* 81(2):91–102.

Hwang, M. H. 2007. *DEVS++ Open Source*. http://odevspp.sourceforge.net/.

Lee, E. A., J. Liu, L. Muliadi, and H. Zheng. 2004. "Discrete-Event Models". In *System Design, Modeling, and Simulation using Ptolemy II*, Chapter 7, 232–273. Oxford University Press.

Lidwell, W., K. Holden, and J. Butler. 2010. *Universal Principles of Design*. Beverly: Rockport Publishers.

Maleki, M., R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. 2015. "Designing DEVS Visual Interfaces for End-User Programmers". *Simulation* 91(8):715–734.

Muzy, A., and B. P. Zeigler. 2014. "Specification of Dynamic Structure Discrete Event Systems using Single Point Encapsulated Control Functions". *International Journal of Modeling, Simulation, and Scientific Computing* 5(3).

Quesnel, G., R. Duboz, and E. Ramat. 2009. "The Virtual Laboratory Environment – An Operational Framework for Multi-Modelling, Simulation and Analysis of Complex Dynamical Systems". *Simulation Modelling Practice and Theory* 17(4):641–653.

Sarjoughian, H. S., and V. Elamvazhuthi. 2009. "CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation". In *Proceedings of the Simulation Tools and Techniques Conference*, SIMUTools '09, 9 pages. Rome, Italy.

Seo, C., B. P. Zeigler, R. Coop, and D. Kim. 2013. "DEVS Modeling and Simulation Methodology with MS4 Me Software Tool". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '13, 8 pages. San Diego, California.

Steiniger, A., F. Krüger, and A. M. Uhrmacher. 2012. "Modeling Agents and their Environment in Multi-Level-DEVS". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque et al., 2978–2989. Piscataway, New Jersey: IEEE.

Steiniger, A., and A. M. Uhrmacher. 2016. "Intensional Couplings in Variable-Structure Models: An Exploration Based on Multi-Level-DEVS". *ACM Transactions on Modeling and Computer Simulation* 26(2).

Vangheluwe, H. L. M. 2000. "DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling". In *Proceedings of the IEEE Symposium on Computer-Aided Control System Design*, CACSD '00, 6 pages. Anchorage, Alaska.

Varga, A., and R. Hornig. 2008. "An Overview of the OMNeT++ Simulation Environment". In *Proceedings of the Simulation Tools and Techniques Conference*, SIMUTools '08, 10 pages. Marseille, France.

Yilmaz, L., and T. I. Ören. 2009. "Agent-Directed Simulation". In *Agent-Directed Simulation and Systems Engineering*, Chapter 4, 111–143. Weinheim: Wiley-VCH.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Second ed. San Diego: Academic Press.

## AUTHOR BIOGRAPHIES

**RHYS GOLDSTEIN** is a research scientist at Autodesk Research specializing in simulation theory and its application in systems modeling and architectural design. He served as a co-chair of the SimAUD 2014/2015 and the TMS/DEVS 2017 symposia, and as an associate editor of *Simulation: Transactions of the Society for Modeling and Simulation International*. His e-mail address is rhys.goldstein@autodesk.com.

**SIMON BRESLAV** is a research scientist at Autodesk Research specializing in computer graphics, information visualization, and human-computer interaction. His research projects include interactive systems for exploring data from sensor networks and simulation. His e-mail address is simon.breslav@autodesk.com.

**AZAM KHAN** is Director, Complex Systems Research at Autodesk specializing in simulation, human-computer interaction, architectural design, and sustainability. He is the founder of the Parametric Human Project Consortium, the SimAUD Symposium, and the CHI Sustainability Community. Azam holds a Ph.D. in computer science from the University of Copenhagen. His e-mail address is azam.khan@autodesk.com.