Informal DEVS Conventions Motivated by Practical Considerations

Rhys Goldstein, Simon Breslav, Azam Khan
Autodesk Research
210 King Street East, Toronto, ON, Canada
{**firstname.lastname@autodesk.com**}

Keywords: simulation tools, convenience, efficiency, reproducibility, encapsulation

Abstract

The formalism known as the Discrete Event System Specification (DEVS) provides a set of mathematical elements for modeling time-varying systems. When DEVS is applied in the form of an executable representation, however, some deviation from the formalism is unavoidable. By proposing a set of informal DEVS conventions, we show how certain changes to the formalism, some previously adopted, others less explored, may help simulation tools appeal to users who stand to benefit from DEVS theory but are more cognizant of practical issues. Our conventions use parameters and statistics to encapsulate the state of an atomic model and the composition of a coupled model. They also include changes both to transition functions and the ordering of simultaneous events to promote convenience, efficiency, and reproducibility.

1. INTRODUCTION

Developers of general-purpose simulation software must choose modeling conventions that allow users to implement, integrate, and test models for a wide range of domains including physics, chemistry, biology, economics, social science, engineering, and architecture. The Discrete Event System Specification (DEVS) has to a large extent addressed this challenge by providing a way to represent essentially any time-varying system as an indivisible atomic model, or as a coupled model defining a network of components that are themselves described by atomic or coupled models (Zeigler et al., 2000).

DEVS can be applied in either of two forms: as a formalism for specifying models using mathematical notation, or as an executable representation for implementing models using a programming language. When the formalism is used, atomic model specifications with the elements $\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$ and coupled model specifications with the elements $\langle X, Y, D, \{M_d : d \in D\}, EIC, EOC, IC, Select \rangle$ can be analyzed using the mathematical laws that constitute DEVS theory. When an executable representation is used, atomic and coupled models can be interpreted by DEVS-based tools to direct computer simulations. Most executable representations are tool-specific, though efforts are underway to define a common standard (Wainer et al., 2010).

All executable DEVS representations differ in various ways from the underlying formalism. Some of these differences are a matter of necessity. For example, atomic model specifications include three purely mathematical sets that cannot be represented in a typical programming language. Most changes, however, are made for the sake of convenience, efficiency, reproducibility, or some other practical benefit. Examples include the merging of two atomic model functions to improve efficiency, or the simplification of an inconvenient coupled model function that orders simultaneous events. These deviations from the formalism, and the practical considerations that motivate them, are the focus of this paper.

Our primary purpose is to provide a set of informal DEVS conventions for atomic and coupled models to aid in the development of simulation tools intended for a particular type of user. The user we consider is one who is an expert in his or her domain, but has little familiarity with DEVS or any other modeling formalism. If such users perceive, rightly or wrongly, that any aspect of DEVS has unnecessarily inconvenienced them, led to inefficient code, or rendered their results irreproducible, they are unlikely to regard consistency with DEVS theory as adequate compensation. Nevertheless, it is our belief that these domain experts have the most to gain from adopting DEVS. We strive for conventions that are sufficiently consistent with the formalism that its main advantage, a more scalable and collaborative approach to simulation, can be realized. However, we deviate from the formalism where necessary to provide at least one solution to each of the dozen or so practical issues we identify herein.

The proposed conventions are partly influenced by changes to the formalism found in existing DEVS-based tools; this paper collects a selection of these changes for the benefit of future tool developers. The conventions also reflect several ideas that emerged through the development of our own DEVS-based software. In presenting them, we compare the proposed elements of atomic and coupled models to those defined in the formalism. Particular attention is given to the incorporation of parameters and statistics, modifications to the transition functions, and the ordering of simultaneous events.

2. ATOMIC MODEL CONVENTIONS

Table 1 lists the elements of atomic models as defined by the DEVS formalism. The atomic model elements we propose are listed in Table 2.

Table 1. Atomic Model Formalism Elements

Element	Description
Input Set (X)	Set of all possible inputs
Output Set (Y)	Set of all possible outputs
State Set (S)	Set of all possible states
Time Advance ($ta : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$)*	Function that... ...takes the current state ...results in the time delay
External Transition ($\delta_{ext} : Q \times X \rightarrow S$)**	Function that... ...takes the current state ...takes the elapsed time ...takes an input value ...results in the new state
Output ($\lambda : S \rightarrow Y \cup \{\varnothing\}$)	Function that... ...takes the current state ...results in an output value
Internal Transition ($\delta_{int} : S \rightarrow S$)	Function that... ...takes the current state ...results in the new state

* $\mathbb{R}_0^+$ is the set of non-negative real numbers
** $Q = \{(s, \Delta t_e) \in S \times (\mathbb{R}_0^+ \cup \{\infty\}) : \Delta t_e < ta(s)\}$

Table 2. Atomic Model Proposed Elements

Element	Description
Model Description	User-supplied documentation
Input Port List	List of input port names
Output Port List	List of output port names
Parameter List	List of parameter names
Statistic List	List of statistic names
Constant List	List of constant names
State Variable List	List of state variable names
Constant Initialization	Function that... ...reads parameter values ...initializes constants ...acquires computer resources
State Initialization	Function that... ...reads constants ...initializes state variables
Time Advance	Function that... ...reads constants ...reads state variables ...provides the time delay
External Transition	Function that... ...reads constants ...reads/modifies state variables ...reads the elapsed time ...reads an input
Internal Transition	Function that... ...reads constants ...reads/modifies state variables ...reads the elapsed time ...provides an output list
Finalization	Function that... ...reads constants ...reads/modifies state variables ...reads the elapsed time ...provides statistic values ...releases computer resources

Starting from the top of Table 2, the Model Description element serves as a reminder that users of DEVS-based tools should be able to incorporate documentation into their models. Note that because our conventions do not constitute an executable representation, the form of each element is left open to interpretation. For DEVS implementations similar to PythonDEVS (Bolduc and Vangheluwe, 2002), which is a library of DEVS-related classes one imports into a Python program, the Model Description could take the form of commented-out lines in source code. For tools such as PowerDEVS (Bergero and Kofman, 2011), which provides graphical user interfaces for model development, it could take the form of a text field, or perhaps a combination of marked-up text and images.

The Input Set (X) and Output Set (Y) have been replaced with Input and Output Port Lists, which is common practice since purely mathematical sets are not easily exploited in typical programming languages. PythonDEVS, for example, provides `addInPort` and `addOutPort` methods to construct these lists. It is important to note that port lists address only a part of X and Y, the other part being the set of values permitted on each port. Here there is an entire spectrum of possible approaches. One can simply ignore the issue and permit any value on any port. One can also assign each port an informal description, or a predicate indicating whether an encountered value is acceptable. Although we include port lists in our conventions, we offer no recommendation on how to restrict the associated values. The reason is that many solutions are only appropriate for certain programming languages. With C++, for instance, one may use templates to associate each model with a datatype for its inputs and outputs (Nutaro, 2011).

The newly added Parameter List, Statistic List, Constant List, the two Initialization functions, and the Finalization function are described in Section 2.1.

The State Set (S) has been replaced by a State Variable List, which is also common practice. This list may take form of variable declarations in a statically typed language such as C, C++, or Java. Alternatively, the list may be implicit in the variable assignment statements of a dynamic programming language such as Python or Lua.

The transition functions have undergone several modifications, an example being the absorption of the Output function into the Internal Transition function. These changes are explained in Section 2.2.

We have not proposed any significant change to the Time Advance function (*ta*). However, we have assumed that modifications to the state variables can be prevented in this function, and that the resulting time delay can be assigned an infinite value. If this is not the case, DEVS-based tool developers should consider absorbing the Time Advance function into the State Initialization function and both External and Internal Transition functions. This is done, for example, in CD++ (Wainer, 2009).

2.1. Atomic Model Parameters and Statistics

Given the popularity of object-oriented programming, domain experts are likely to understand encapsulation even if they are unfamiliar with DEVS-specific principles. The Input and Output Port Lists encapsulate the state variables of an atomic model during a simulation; from outside of the model, its state can be only be influenced or accessed indirectly via inputs and outputs. Several of the proposed elements are useful, however, to ensure the state variables are also encapsulated at the beginning and end of a simulation.

At the beginning of a simulation, the user can supply values for the parameters declared in the Parameter List. The model receives these parameter values, and uses them to calculate the initial values of the state variables. By allowing the initial state of an atomic model to be configured indirectly via parameters, the state variables remain encapsulated. This convention is by no means original. Most object-oriented DEVS libraries feature initialization methods or class constructors that read parameter values.

Parameters not only control the initial state of an atomic model, they also customize its subsequent behavior by influencing the model's functions. Having all functions read parameter values directly could be inefficient, however, as the same parameter-dependent calculations might be required in multiple functions or in repeated invocations of the same function. Thus it is for the sake of efficiency that we propose a Constant List and the use of two initialization functions instead of one. The Constant Initialization function is invoked first. It is the only function in which parameter values can be accessed and constants can be initialized and modified. The ability to modify constants in this function is important for populating arrays and data structures that will later be immutable. The State Initialization function is separated from the first initialization function so that modifications to the constants can be disallowed before the state variables acquire their initial values. Note that if a modeler desires direct access to the parameter values in every atomic model function, they need only define a one-to-one mapping between parameters and constants.

At the end of a simulation, we propose that a Finalization function be invoked to supply a value for each statistic declared in the Statistic List. This is preferable to directly exposing the final values of the state variables, which would violate the principle of encapsulation. It should be noted that the Finalization function, which may also be used to release computer resources acquired during initialization, is not an original concept. The Exit function in PowerDEVS, for instance, is similar. However, because statistics are usually derived from a model's outputs, our suggestion to incorporate them into the model is a departure from common practice.

To understand the rationale for calculating statistics within a model, consider a hypothetical simulation designed to integrate a differential equation between predefined start and end times. The purpose of the simulation is to report on statistics that depend on the result of this integration. The integration is performed by a model that outputs its progress at every internal transition. A problem arises if the simulation ends between two internal transitions, as there will be no event to calculate and deliver the final segment of the integration results. In this case, the Finalization function provides a place to complete the integration. To fulfill this task, the function must have access to the time elapsed since the previous transition. Also, in order to reuse numerical integration code, the function must be able to modify the state variables. Finally, since there is no output to deliver the result of the integration, the model itself must provide the required information in the form of statistics.

There are many cases where simulation results should be fully derived from models outputs; in those cases, modelers can leave the Statistic List empty and the Finalization function blank. Nevertheless, we believe these elements should be included in DEVS-based software to give users an additional option for scenarios such as the one described above.

2.2. Transition Functions

As mentioned earlier, our conventions involve absorbing the Output function (λ) into the Internal Transition function (δ_{int}). In other words, the Output function is omitted, while the Internal Transition function acquires the option of producing an output. This is essentially the convention adopted by DEVS++ (Hwang, 2009). The motivation for merging the two functions is the observation that λ and δ_{int} often involve same intermediate calculations. If the functions are kept separate, as they are in most existing DEVS-based tools, these calculations must be performed twice instead of once. Alternatively, the software can allow the Output function to make state changes, but this also contradicts DEVS theory and may confuse users who realize that they can move code between the two functions without changing their results.

There is some theoretical justification for merging λ and δ_{int}. Let $\hat{\delta}_{int} : S \to S \times Y$ be the resulting function. Given any atomic model specification, we can derive $\hat{\delta}_{int}$ using (1).

$$\hat{\delta}_{int}(s) = (\delta_{int}(s), \lambda(s)) \tag{1}$$

Also, although we do not recommend using our conventions for specification purposes, (2) demonstrates that there is a mapping from $\hat{\delta}_{int}$ back to λ and δ_{int}.

$$\begin{aligned} \delta_{int}(s) &= s' \quad \text{where } (s', y) = \hat{\delta}_{int}(s) \\ \lambda(s) &= y \quad \text{where } (s', y) = \hat{\delta}_{int}(s) \end{aligned} \tag{2}$$

Finally, since λ may take on the form shown in (3), the DEVS formalism does permit the output to depend on the post-transition state instead of the pre-transition state.

$$\lambda(s) = f(\delta_{int}(s)) \quad \text{for any function } f \tag{3}$$

Despite these justifications, the decision to merge λ and δ_{int} should not be made lightly. One drawback is that it complicates efforts to prove that an atomic model is formally legitimate. Legitimacy means that in the absence of inputs, simulated time necessarily advances towards ∞ without stopping or converging (Zeigler et al., 2000). Note that the formula for legitimacy, shown below, depends on δ_{int} but not λ.

$$\sum_{i=0}^{\infty} ta(s_i) = \infty \quad \text{for all } s_0 \in S \text{ where for } i > 0,\ s_i = \delta_{int}(s_{i-1})$$

By merging λ and δ_{int}, it is nearly certain that some or all of the Stochastic DEVS (STDEVS) theory described in Castro et al. (2008) will be rendered inapplicable. The reason is simply that (1), (2), and (3) all assume atomic model functions are deterministic. One may, of course, still sample a random number generator in our proposed Internal Transition function, so it is unclear whether inconsistency with STDEVS will impact domain experts. We expect such users to be more concerned about the loss of efficiency associated with keeping the functions separate.

A third drawback to merging λ and δ_{int} is that it is not an option for Parallel DEVS, a variant of DEVS that exploits parallel computing technology by dispensing with the ordering of simultaneous events (Chow and Zeigler, 1994). In general, the solutions we propose are applicable to Classic DEVS, not its variants, though one could argue that an ideal set of conventions would take numerous variants into account. In Parallel DEVS, an invocation of λ is only sometimes followed by δ_{int}, so combining the functions is complicated at best. The problem of avoiding redundant calculations still exists, however, and is worthy of attention.

Instead of producing one optional output, as done by λ, our Internal Transition function produces a list. This list of outputs, motivated by convenience, is comparable to the bag of outputs found in software implementing Parallel DEVS. In DEVSJava (Zeigler and Sarjoughian, 2005), for example, a model may use repeated calls of the form `makeContent(port, value)` to produce simultaneous outputs. The difference in our case is that the outputs are propagated not simultaneously, but in the order they are listed. We will discuss this further in the context of coupled models.

If our Internal Transition function is to provide a list of outputs instead of one optional output, it seems intuitive that the External Transition function (δ_{ext}) take a list of inputs instead of a single input. But other than symmetry, we do not see any practical benefit to altering δ_{ext} in this way. In most cases, receiving inputs one at a time conveniently alleviates the need to iterate over a list. In cases where a list of inputs is necessary, it is not difficult to have δ_{ext} queue incoming values for subsequent processing.

Observe in Tables 1 and 2 that, instead of "taking the current state" and "resulting in the new state", our transition functions "read/modify the state variables". The difference in phrasing becomes important if a model's state requires a considerable amount of memory. If the current state is immutable, as is the case with the SC-DEVS tool (Madlener et al., 2009), then extra memory will need to be allocated for the new state, and a significant amount of data is likely to be copied at every transition. Most DEVS-based tools allow the state variables to be modified, which we recommend for the sake of efficiency.

Finally, observe in Table 2 that the elapsed time is to be made available in the Internal Transition function as well as the External Transition function. This is consistent with the formalism in the sense that, although the elapsed time is not an argument of δ_{int}, it can be obtained via the expression $ta(s)$. In a DEVS-based tool, it may not be practical to invoke the Time Advance function inside the Internal Transition function, so we recommend making the elapsed time available by another mechanism.

3. COUPLED MODEL CONVENTIONS

The elements of coupled models, as defined by the formalism, are listed in Table 3. The elements we propose are listed in Table 4.

Observe that the first two elements in Table 3 are identical to those previously encountered in Table 1. These elements constitute the interface to any DEVS model according to the formalism. Likewise, the first five elements of Table 4 are identical to those in Table 2, as these elements constitute the proposed model interface. The main difference is that the proposed interface accounts for parameters and statistics. Although the sixth element, the Constant List, is also found in both atomic and coupled models, it is part of the implementation of a model. The parameters, statistics, and constants of coupled models, as well as their Initialization and Finalization functions, are discussed in Section 3.1.

The proposed elements that represent the composition of a model, the Component List, the Component Models, and the Coupling, are similar to the corresponding elements of the formalism $(D, \{M_d\}, EIC, EOC, IC)$. The one notable differ-

Table 3. Coupled Model Formalism Elements

Element	Description
Input Set (X)	Set of all possible inputs
Output Set (Y)	Set of all possible outputs
Component Set (D)	Set of component names
Component Models ($\{M_d : d \in D\}$)	Atomic and/or coupled models (one model per component)
External Input Coupling (EIC)	Set of links connecting... ...input ports to components
External Output Coupling (EOC)	Set of links connecting... ...components to output ports
Internal Coupling (IC)	Set of links connecting ... ...components to components
Tie-Breaking ($Select : 2^D \rightarrow D$)	Function that... ...takes a set of component names ...results in a selected name

Table 4. Coupled Model Proposed Elements

Element	Description
Model Description	User-supplied documentation
Input Port List	List of input port names
Output Port List	List of output port names
Parameter List	List of parameter names
Statistic List	List of statistic names
Constant List	List of constant names
Component List	List of component names
Component Models	Mapping that... ...takes any component name ...identifies the model that describes the component
Coupling	Set of links connecting... ...input ports to components ...components to output ports ...components to components
Initialization	Function that... ...reads parameter values ...initializes constants ...provides parameter values to components ...acquires computer resources
Finalization	Function that... ...reads constants ...reads the elapsed time ...reads the total elapsed time ...reads the remaining time ...reads statistic values from components ...provides statistic values ...releases computer resources

ence is that, if a proposed element is described as a *list* instead of a *set*, the user is to have control over the order of its items. Again, we emphasize that the form of each proposed element is open to interpretation. The Component List need not be presented to the user as a list of names. For our own software, we adopted the well-established approach of representing components as labeled nodes in an editable diagram. The nodes are numbered to indicate the order of the components. For the majority of the proposed list elements, the fact that the items are ordered allows them to be presented in a consistent fashion. Some of these orderings also affect the execution of simultaneous events, as explained in Section 3.2.

3.1. Coupled Model Parameters and Statistics

A simple way to define the Parameter Lists of coupled models is to always concatenate the Parameter Lists of their components. Unfortunately, this violates the principle of encapsulation by exposing the composition of a coupled model through its interface. It can also be impractical, as a simple example illustrates. Consider a coupled model with five components. Each component has a parameter named *color*, and the coupled model is valid only if all five components are initialized with the same value for this parameter. If the parameter lists are concatenated, the coupled model ends up with five separate *color* parameters that require the same value, which is both inconvenient and error-prone.

Our conventions require a coupled model's parameters to be listed independently of those of its components. At the beginning of a simulation, the Initialization function reads the coupled model's parameter values and computes a separate list of parameter values for each component. Next, the Initialization function of each component is invoked with the appropriate one of these computed lists. Note that if the component is described by an atomic model, it is the Constant Initialization function that is invoked here, not the State Initialization function. This overall approach minimizes the number of parameters required by a coupled model, and allows the model to automatically enforce any necessary constraints involving the parameters of its components.

A similar argument can be made against routinely concatenating the Statistic Lists of the components of a coupled model. We require a coupled model's statistics to be listed independently. At the end of a simulation run, the Finalization function of each component computes a list of statistic values. Next, the coupled model's Finalization function is invoked to read all of these computed lists and produce its own statistic values. The Finalization function of a coupled model may read parameter-dependent constants populated in the Initialization function. It may also read the time elapsed since the previous component-level event, the total time elapsed since the beginning of the simulation, and the time remaining until the next component-level internal transition.

3.2. Simultaneous Events

When performing simulations that are neither parallelized nor interactive, domain experts may justifiably expect their results to be reproducible. To meet this expectation, conventions must be chosen to deterministically order simultaneous events that arise in four different situations.

The first situation occurs when multiple components are scheduled to undergo internal transitions at the same time. According to the formalism, the Tie-Breaking function (*Select*) is invoked to choose one of these components. Because *Select* is inconvenient to define in its general form, which accounts for all subsets of a coupled model's components, our conventions omit it. The ordering required by the Component List is used lieu of *Select* to determine the next internal transition. CD++ (Wainer, 2009) is an example of an existing DEVS-based tool that uses this convention.

The second situation results from allowing a single internal transition to produce a list of outputs. Each output triggers a separate external transition, so these transitions must be ordered. Since the outputs are themselves ordered, the same ordering can be applied to the external transitions.

The third situation arises when a single output is received by more that one component. In theory, the order of the external transitions triggered by this output has no effect on any subsequent output, so the order is left unspecified. In practice, because multiple components may draw samples from the same pseudorandom number generator, changing the order of these external transitions can alter simulation results. Suppose that such orderings are determined by the sequence in which links between ports are drawn in a graphical editor. In that case, two modelers can create models that appear identical, yet produce different results when simulated under exactly the same conditions. To promote reproducibility when it is expected, we recommend that simultaneous external transitions involving multiple components be ordered in the same manner as internal transitions: using the Component List.

The fourth situation pertains to a single output received by a component on multiple ports. In this case, the order of the resulting external transitions can affect subsequent outputs even in the absence of pseudorandom numbers or other side effects. To address the same reproducibility issue described above, the ordering required by the Input Port List of the receiving component can be applied to its external transitions.

4. FUTURE WORK

This work could be extended to provide informal conventions for DEVS-based experiments, including uncertainty, sensitivity, and parameter optimization analyses requiring multiple simulation runs. By allowing coupled models to enforce relationships between component parameters, and by providing statistics that can be used to implement cost functions, the conventions proposed here support such efforts.

We encourage others to propose analogous conventions for Parallel DEVS and other DEVS variants. Although our solutions do not apply to these variants, many of the practical issues remain relevant. Addressing them will make DEVS-based tools compelling to a broader range of potential users.

REFERENCES

Bergero, F. and E. Kofman (2011). PowerDEVS: A Tool for Hybrid System Modeling and Real-Time Simulation. *Simulation 87*(1-2), 113–132.

Bolduc, J.-S. and H. Vangheluwe (2002). A Modeling and Simulation Package for Classic Hierarchical DEVS. Technical report, School of Computer Science, McGill University.

Castro, R., E. Kofman, and G. Wainer (2008). A Formal Framework for Stochastic DEVS Modeling and Simulation. In *Proceedings of the Spring Simulation Conference*.

Chow, A. C. H. and B. P. Zeigler (1994). Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism. In *Proceedings of the Winter Simulation Conference*.

Hwang, M. H. (2009). *DEVS++: C++ Open Source Library of DEVS Formalism* (v.1.4.2 ed.).

Madlener, F., H. G. Molter, and H. Sorin A (2009). SC-DEVS: An efficient SystemC Extension for the DEVS Model of Computation. In *Proceedings of the Design, Automation, and Test in Europe Conference*.

Nutaro, J. J. (2011). *Building Software for Simulation: Theory and Algorithms with Applications in C++*. Hoboken, NJ, USA: John Wiley & Sons.

Wainer, G. A. (2009). *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL, USA: CRC Press.

Wainer, G. A., K. Al-Zoubi, D. R. C. Hill, S. Mittal, J. L. R. Martín, H. Sarjoughian, L. Touraille, M. K. Traoré, and B. P. Zeigler (2010). An Introduction to DEVS Standardization. In G. A. Wainer and P. J. Mosterman (Eds.), *Discrete-Event Modeling and Simulation: Theory and Applications*, Boca Raton, FL, USA: CRC Press, pp. 393–425.

Zeigler, B. P., H. Praehofer, and T. G. Kim (2000). *Theory of Modeling and Simulation* (2nd ed.). San Diego, CA, USA: Academic Press.

Zeigler, B. P. and H. S. Sarjoughian (2005). Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models. Technical report, Arizona Center for Integrative Modeling and Simulation, University of Arizona.