Implementation-Oriented DEVS Conventions

Name of Author(s)
And Affiliations(s)
(Use Upper and Lower Case)
Include email address, preferably as a live hyperlink
If multiple author affiliations, you can use multiple columns

Keywords: simulation tools, time resolution, encapsulation

Abstract

The mathematical modeling conventions known collectively as the Discrete Event System Specification (DEVS) can be used to specify models of essentially any system that varies in time. But when DEVS is used for implementation, as opposed to specification, practical considerations such as convenience, efficiency, and repeatability motivate various changes to the original conventions. Here we discuss a selection of these changes found in existing DEVS-based software. We then provide a set of implementation-oriented DEVS conventions to guide the development of future simulation tools. The proposed conventions comply with DEVS principles by improving a user's control over the timing of events, and supporting the encapsulation of a model's state or composition.

1. INTRODUCTION

Developers of general-purpose simulation tools strive to choose reasonably small sets of modeling conventions that allow users to implement, integrate, and test a wide range of models for different applications. 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 atomic model consisting of seven elements: three mathematical sets and four pure functions (Zeigler et al., 2000). A system can also be represented as a coupled model defining a network of other models.

True to its name, DEVS in its original form is meant for specifying, not implementing, atomic and coupled models. It is therefore unsurprising that when DEVS is used for implementation, various practical considerations require tool developers to deviate from the original conventions. Some of these deviations are a matter of necessity. For example, atomic model specifications include three purely mathematical sets that cannot be represented in a typical programming language. But most changes are made for the sake of convenience, efficiency, repeatability, or other practical benefits. Examples include the merging of two atomic model functions to improve efficiency, and the omission of an inconvenient coupled model function that orders simultaneous events. Deviations like these can be found in all existing DEVS-based tools; a selection of them are described in Section 2.

Our main purpose here is to provide a set of DEVS conventions for implementing atomic and coupled models, as well as configuring and reporting on simulation runs. These implementation-oriented DEVS conventions may guide the development of future simulation tools exhibiting a range of programming languages and types of user interfaces. The proposed conventions are listed, and compared with the original specification-oriented conventions, in Section 3.

Counterintuitively, the use of an alternative set of DEVS conventions may help simulation tools comply with the underlying principles of DEVS. For example, although a floating-point time representation seems consistent with the original conventions, an integer-based representation reduces round-off errors and gives simulation software users greater control over the timing of events. By requiring a time resolution for every model and simulation run, our conventions support integer-based time values over a wide range of time scales. Time resolution is discussed in Section 4.

We also discuss the principle of encapsulation. In theory, the state of an atomic model and the composition of a coupled model can be considered encapsulated, as neither need be referenced by a simulation run or another coupled model. What complicates matters in practice is the need to control a model's initial state and extract information from its final state without exposing any state variables. Parameters, statistics, constants, and initialization and finalization functions help address this issue, as explained in Section 5.

2. DEVS CONVENTIONS IN PRACTICE

There are many different types of DEVS-based simulation tools. PythonDEVS (Bolduc and Vangheluwe, 2002) is simply a library of DEVS-related classes one imports into their own Python program. PowerDEVS (Bergero and Kofman, 2011), by contrast, comprises several graphical user interfaces for coding atomic models in C++, editing coupled models visually, and running simulations. A number of implementation-related issues involving the original DEVS conventions arise in all such tools, regardless of the type of user interface or the supported programming language. Here we describe these issues and observe how they are handled in existing DEVS-based tools. We also point out tested approaches that have influenced the implementation-oriented DEVS conventions proposed in Section 3.

In the original DEVS conventions, both atomic and coupled models include sets X and Y to define the possible inputs and outputs. Since mathematical sets cannot be represented in a typical programming language, it is common practice to replace them with lists of input and output ports. PythonDEVS, for example, provides `addInPort` and `addOutPort` methods to construct these lists.

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 implementation-oriented 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).

Turning our attention from sets to functions, consider the following internal transition function δ_{int} of an atomic model. The function takes a model's current state, s, and yields the new state, in this case $s+1$.

$$\delta_{int}(s) = s+1$$

When such a function is implemented, the question arises as to whether the current state and new state should share memory. The C/C++ code below illustrates both options.

```
state delta_int(const state& s)
  { return s+1; }

void delta_int(state& s)
  { s = s+1; }
```

The first version of `delta_int` is more consistent with the original DEVS conventions, and mathematical notation in general, since the current state `s` cannot be modified and the new state `s+1` is a distinct value. However, by allowing the current state to be modified to produce the new state, the approach demonstrated in the second version reduces memory use and alleviates the need to populate the entire state if only a small part of it is to change. The first approach is used in SC-DEVS (Madlener et al., 2009), but the majority of existing DEVS-based tools opt for efficiency and allow modifications to one or more state variables. In our conventions, we describe both internal and external transition functions as modifying, not replacing, the state.

The internal transition function δ_{int} mentioned above is always invoked after the output function λ, and as demonstrated by DEVS++ (Hwang, 2009) it is possible to merge the two functions into one. The motivation for this change is the observation that λ and δ_{int} often involve same intermediate calculations. If the two functions are kept separate, as they are in most existing DEVS-based tools, these calculations must be performed twice instead of once. Alternatively, the intermediate results may be stored by λ and retrieved by δ_{int}, but permitting what is essentially a state change in λ would render δ_{int} unnecessary. Our conventions omit λ and allow δ_{int} to provide an output in addition to changing the state. Actually, as explained shortly, our convention is that the internal transition function provide not one optional output, but a list of zero or more outputs.

It is worth noting that the changes we propose are not, in general, applicable to the many mathematical variants of DEVS. Consider Parallel DEVS (Chow and Zeigler, 1994), which exploits parallel computing technology by dispensing with the ordering of simultaneous events. In Parallel DEVS, an invocation of λ is only sometimes followed by δ_{int}, so merging the two functions would be complicated at best.

A set of implementation-oriented conventions for Parallel DEVS is beyond the scope of this paper. However, the list of outputs provided by our version of the internal transition function is somewhat similar 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 in the order they are listed. If a model produces two outputs and the first is to be received by another component, the recipient's external transition function δ_{ext} is invoked before the second output is propagated.

If δ_{int} is to provide a list of outputs instead of one optional output, it seems intuitive that 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.

Turning our attention from atomic models to coupled models, the tie-breaking function *Select* determines which component first undergoes an internal transition should there be a tie. In CD++ (Wainer, 2009), the order of simultaneous internal transitions adheres to the order in which the component names are listed, rendering *Select* unnecessary. We adopt this convention for the sake of convenience, and use the same ordering of components to determine which external transition occurs first should multiple components receive an input from a common source. The order of such external transitions is irrelevant from a mathematical perspective. But from a technological point of view, the presence of side effects and pseudorandom numbers, combined with the importance of repeatable simulation runs, motivates a deterministic order for all events. Ordering components is a convenient way to break ties for both internal and external transitions.

3. PROPOSED CONVENTIONS

The six tables presented in this section list the elements of the original, specification-oriented DEVS conventions and the proposed, implementation-oriented DEVS conventions for atomic models, coupled models, and simulation runs.

3.1. Atomic Model Conventions

The original atomic model conventions in Table 1 and the proposed conventions in Table 2 include a number of corresponding elements. The Input Set X and Output Set Y have been replaced with Input and Output Port Lists, as was mentioned in Section 2. We did not yet mention that the State Set S would be replaced by a State Variable List, but it is common practice to represent a model's state with multiple variables. Observe that each implementation-oriented transition function "reads" and "modifies" these state variables, as we assume current states and new states share memory. Aside from that difference, the external transition function is similar in both sets of conventions. The time advance function is similar as well, but the proposed internal transition function absorbs the output function by providing an output list.

Table 1. Specification-Oriented Atomic Models

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

** $Q = \{(s, \Delta t_e) \in S \times (\mathbb{R}_0^+ \cup \{\infty\}) : \Delta t_e < ta(s)\}$

Several new elements are included in the implementation-oriented conventions of Table 2. Starting from the top, it is standard practice that every model have some sort of ID, whether it be a filename or the name of an object-oriented class. We include the ID for the sake of completeness. Also deserving of inclusion is a model's informal description, be it a set of comments associated with an object-oriented class, a text field in a graphical user interface, or an elaborate combination of marked-up text and diagrams. The time resolution element is explained in detail in Section 4. The parameter, statistic, and constant lists, as well as both initialization functions and the finalization function, help encapsulate the state of an atomic model. They are described in Section 5.

Table 2. Implementation-Oriented Atomic Models

Element	Description
Model ID	Model name or identifier
Description	Informal model description
Time Resolution	Bound on time value resolution
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

The original conventions are explicit about the form of each element. For example, the transition functions conform with the equations $s' = \delta_{ext}(s, \Delta t_e, x)$ and $s' = \delta_{int}(s)$, where s' is the new state, s is the current state, Δt_e is the time elapsed since the previous event, and x is the input. In the proposed conventions, the form of each element is considerably more open to interpretation. This is done to accommodate different

types of programming languages and user interfaces. With an object-oriented language, for example, the state variables that constitute s would probably not be provided as arguments to the transition functions. Instead they would be accessed as member variables of a class that incorporates the transition functions as methods.

Consider another example of how the form of each proposed element is open to interpretation. In the original conventions it is clear that δ_{ext} takes the elapsed time Δt_e as an argument, whereas δ_{int} does not. One may still reference the elapsed time in δ_{int}, but this is done using the expression $ta(s)$. Instead of using an argument in one transition and a function call with potential side effects in the other, we recommend that DEVS-based software provide a single mechanism to read the elapsed time in both transition functions as well as in the new finalization function. The form of each of these functions depends on whether the elapsed time is supplied by an argument, by a built-in function, or by some other mechanism.

3.2. Coupled Model Conventions

Shown in Tables 3 and 4 are the original and proposed conventions for coupled models. A number of elements in these tables can be considered part of the interface to any model, be it atomic or coupled. The specification-oriented model interface consists of the first two elements in Tables 1 and 3 (X and Y), while the implementation-oriented model interface consists of the first seven elements in Tables 2 and 4 (Model ID, Description, ..., Statistic List). Simulation tool developers should exploit the fact that these seven implementation-oriented elements are common to all models.

Table 3. Specification-Oriented Coupled Models

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

Let us now clarify the terns "set" and "list" in the context of our proposed conventions for both atomic and coupled models. When we describe an implementation-oriented element as a list, we are recommending that the software user be given control over the order of the items. Admittedly, the ordering of output ports, parameters, statistics, constants, and state variables should have no effect on a simulation run. The main reason for these items to be ordered is to promote consistency throughout a simulation tool; if `length` appears before `width` in one part of a user interface, `width` ought not precede `length` in a different part of the interface. There are two lists, however, that do impact simulation runs according to our conventions. The list of component names in Table 4 implies an ordering of components that may take the place of the *Select* function, as explained in Section 2. Recall that *Select* only orders internal transitions, whereas we would use the order of components to also select the first external transition when multiple components receive an input from a common source. By ordering simultaneous events of either type,

Table 4. Implementation-Oriented Coupled Models

Element	Description
Model ID	Model name or identifier
Description	Informal model description
Time Resolution	Bound on time value resolution
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 ...provides the model ID associated with 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

our policy helps ensure that simulation runs are repeatable. However, there is one more source of simultaneous events to consider. It is rare but possible for multiple ports of a single component to receive an input from a common source. In that case, we recommend that the ordering implied by the receiving model's list of input ports be used to break the tie.

The distinction between sets and lists may affect the design of a simulation tool's user interface. Given a purely textual interface, one can argue that any set of items has at least one implied ordering: the order in which each item appears. However, many DEVS-based tools provide a visual editor for coupled models. Users prepare diagrams of ports, components, and couplings, and in these diagrams the ordering of items may be ambiguous. To comply with our conventions, developers of visual coupled model editors must provide some mechanism for users to order ports and components. The couplings need not be ordered, as they are described as sets in both Tables 3 and 4.

The proposed constant list, initialization function, and finalization function serve a similar purpose for coupled models as the similarly named elements do for atomic models, except that they help encapsulate composition-related information instead of state. This is explained in Section 5.

3.3. Simulation Run Conventions

While the original modeling conventions have been described in numerous books and papers on DEVS, less attention has been paid to the additional elements required to configure and report on simulation runs. Yet for every model, one must be able to record multiple simulation configurations and the associated results. Here we compare the plausible set of simulation run conventions in Table 5 to the proposed set of conventions in Table 6.

Table 5. Specification-Oriented Simulation Runs

Element	Description
Model	Model to be simulated
Initial State(s)	Initial state(s) of model (components)
Input Series	List of time values and inputs
Output Series	List of time values and outputs
Final State(s)	Final state(s) of model (components)

We admit that the specification-oriented conventions in Table 5 may not appear elsewhere in the literature. However, we imagine that these are the conventions a simulation tool developer would attempt to adhere to if deviations from the original modeling conventions were somehow disallowed. The first three elements constitute the simulation configuration, and first among them is the model to be simulated. If the model is atomic, the second element is its initial state. If the model is coupled, the second element is a mathematical structure that associates every component of the model, including components nested within other components, with its own initial state. The third and last configuration element is the input series. It provides a list of inputs, where each input is paired with the simulated time at which it triggers an external transition. The last two elements constitute the simulation results. The first of these is the output series, which becomes populated with outputs and associated internal transition times. The other is either the final state of an atomic model, or the final states of the components of a coupled model.

Table 6. Implementation-Oriented Simulation Runs

Element	Description
Simulation ID	Simulation name or identifier
Model ID	ID of model to be simulated
Description	Informal simulation description
Time Resolution	Resolution of all time values
Random Seed	Random number generator seed
End Time	Bound on simulated time
Parameter Value List	List of parameter values
Input Series	List of time values and inputs
Output Series	List of time values and outputs
Statistic Value List	List of statistic values

A developer who adopts our implementation-oriented conventions for atomic and coupled models will want to consider Table 6 for their simulation runs. This set of simulation conventions is similar to the specification-oriented list in that it includes an input series and an output series, as well as a reference to the model being simulated. One key difference is the replacement of the initial and final state(s) with lists of parameter and statistic values, which are discussed in Section 5. The output series and the list of statistic values constitute the simulation results, whereas the first eight elements in Table 6 are part of the simulation configuration.

There are five elements in Table 6 we have yet to mention. The ID of the simulation run is important, though a filename would suffice. The informal description would ideally be used to relate the simulation run to a broader experiment. The time resolution is explained in Section 4. The random seed is included so that a single pseudorandom number generator, to be shared by all components, can be seeded at the beginning of a simulation run. Finally, the end time provides a means to terminate a simulation run despite the presence of inputs that have yet to be processed, or scheduled internal transitions that have yet to be reached. Only external and internal transitions which occur strictly prior to the end time are to be executed.

Note that the elements in Table 6 are sufficient for only the most basic type of simulation: a single run, executed by a single thread of computation, for which all configuration data is provided at the outset. Simulation tool developers may include additional elements to accommodate parallelism, interactivity, or experiments involving multiple simulation runs.

4. TIME RESOLUTION

A key advantage of DEVS is the control it gives modelers over the timing of events. Unfortunately, if a simulation tool adopts the intuitive and common approach of representing time values with floating-point numbers, the round-off errors that occur when time values are added and subtracted may shift events slightly forwards or backwards in time. The practical consequences of imprecise event times are threefold. First, they complicate the simulation process, as events scheduled for slightly different times may be treated as simultaneous. Second, they can lead to unexpected conditions, such as an elapsed time Δt_e that is slightly greater than a state variable Δt_r representing the time remaining between the previous event and the next scheduled internal transition. When updating the remaining time, one must consider that $\Delta t_r - \Delta t_e$ may be negative. Third, imprecise event times may annoy users who find themselves unable to implement conceptually simple models, such as one that generates outputs at a fixed time interval of exactly 0.1 time units.

CD++ is unusual among existing DEVS-based tools in that all times values are represented by integers. Actually, time values are represented using an object-oriented class named `Time`, but the class encapsulates an integral number of milliseconds. This approach eliminates round-off errors produced by the addition and subtraction of floating-point time values. On the other hand, having a fixed time resolution may render a simulation tool impractical for models involving extreme time scales. A millisecond resolution may be too coarse for simulations of certain chemical and biological processes, though a considerably finer resolution might lack the range needed for geological or cosmological processes.

Our solution is to have all time values represented with integers encapsulated in a class or custom data type, as is done in CD++. However, to support a wide range of time scales, the resolution common to all time values is specified as part of the configuration of a simulation run. This is the role of the time resolution element in Table 6. As indicated in Tables 2 and Tables 4, each model also has a time resolution, the purpose of which is to ensure that the time resolution of a simulation run is sufficiently fine. We present two approaches for selecting a model's time resolution, starting with a mathematical approach that extends DEVS theory.

We consider a positive rational number Δt_{sim} to be a valid simulation time resolution so long as it divides every duration value Δt_{val} that can possibly separate events during a simulation. Equation (1) defines the notation $\Delta t_{sim} | \Delta t_{val}$, meaning "$\Delta t_{sim}$ divides Δt_{val}" or "Δt_{val} is divisible by Δt_{sim}".

$$\Delta t_{sim} | \Delta t_{val} = \left(\frac{\Delta t_{val}}{\Delta t_{sim}} \in \mathbb{N}_0^+ \cup \{\infty\} \right) \tag{1}$$

Every DEVS model specification M has an optimal time resolution $\mathrm{tr}(M)$, the greatest duration that, if positive, is divisible only by valid simulation time resolutions. A model's optimal time resolution is also the supremum over all finite Δt such that, for $n = 1, 2, 3, \ldots$, a simulation time resolution of $\frac{\Delta t}{n}$ is valid and hence divides any result of the model's time advance function ta. The formal definition of $\mathrm{tr}(M)$ for an atomic model is given by (2), where $S_{\Delta t/n}$ is the set of reachable states subject to the condition that the initial state s_0 and every elapsed time Δt_e satisfy $\frac{\Delta t}{n} | ta(s_0)$ and $\frac{\Delta t}{n} | \Delta t_e$.

$$\mathrm{tr}(M) = \sup_{\Delta t \in \mathbb{Q}^+} \left(\forall n \in \mathbb{N}^+, \forall s \in S_{\Delta t/n}, \frac{\Delta t}{n} \middle| ta(s) \right) \tag{2}$$

One can often identify the optimal time resolution of a model without formally evaluating (2). To demonstrate, we analyze the simple DEVS models specified in Table 7, starting with the Counting Generator. This model outputs an incrementing number every 480 seconds, or 8 minutes, and unsurprisingly this interval turns out to be the optimal time resolution. Looking at the formula, 8 minutes is the greatest Δt in the set of positive rational numbers $\mathbb{Q}^+$ for which $\Delta t, \frac{\Delta t}{2}, \frac{\Delta t}{3}, \ldots$ all divide every possible result of ta. The analysis is trivial since this ta always yields the same duration.

The Simple Processor model is just slightly more complex. It receives the number n as an input, waits 45 seconds or 1 minute depending on whether n is odd or even, then outputs n. Here ta yields ∞, 45, or 60 seconds. According to (1), a duration of ∞ is divisible by any time resolution, so we focus instead on 45 and 60 seconds. The optimal time resolution is 15 seconds, the greatest common divisor of these durations.

The Loyal Processor is similar to the Simple Processor, except that it ignores any input it receives while processing the previous input. In addition to ∞, 45, and 60 seconds, ta may now yield $\Delta t_r - \Delta t_e$, the difference between the previous remaining time and the elapsed time. Note that $\Delta t_r - \Delta t_e$ is not always divisible by 15 seconds. However, $\Delta t_r - \Delta t_e$ is divisible by a fraction of 15 seconds so long as that fraction divides all possible Δt_e. This is all that (2) requires, so $\mathrm{tr}(M)$ is again 15 seconds. In general, because a model has no influence on the Δt_e values it receives, subtracting Δt_e to produce the result of ta has no effect on the model's optimal time resolution.

The Ideal Processor outputs whatever number it receives without delay. Here ta yields either 0 or ∞, which are both divisible by any time resolution according to (1). The optimal time resolution is thus the supremum over all finite Δt, which is ∞, and this means that the model should impose no constraint on the time resolution of a simulation.

The Random Generator is similar to the Counting Generator, except that the time intervals between outputs are randomly sampled from an exponential distribution. The real numbers resulting from ta are not generally divisible by any rational number. Hence $\mathrm{tr}(M)$ is the supremum of an empty set of positive rationale numbers, which we interpret as 0. If a model has an optimal time resolution of 0, it is theoretically impossible to choose a valid simulation time resolution.

Table 7. Simple DEVS Model Specifications

Model	Time Advance	External Transition	Internal Transition
Counting Generator	$ta(n) = 480$	undefined	$\delta_{int}(n) = n+1$
Simple Processor	$ta(n) = \begin{cases} \infty & \text{if } n = \varnothing \\ 45 & \text{if } \frac{n}{2} \notin \mathbb{N} \\ 60 & \text{if } \frac{n}{2} \in \mathbb{N} \end{cases}$	$\delta_{ext}(n, \Delta t_e, x) = x$	$\delta_{int}(n) = \varnothing$
Loyal Processor	$ta(n, \Delta t_r) = \Delta t_r$	$\delta_{ext}(n, \Delta t_r, \Delta t_e, x) = \begin{cases} (x, 45) & \text{if } (n = \varnothing) \wedge (\frac{x}{2} \notin \mathbb{N}) \\ (x, 60) & \text{if } (n = \varnothing) \wedge (\frac{x}{2} \in \mathbb{N}) \\ (n, \Delta t_r - \Delta t_e) & \text{if } n \in \mathbb{N} \end{cases}$	$\delta_{int}(n, \Delta t_r) = (\varnothing, \infty)$
Ideal Processor	$ta(n) = \begin{cases} \infty & \text{if } n = \varnothing \\ 0 & \text{if } n \in \mathbb{N} \end{cases}$	$\delta_{ext}(n, \Delta t_e, x) = x$	$\delta_{int}(n) = \varnothing$
Random Generator	$ta(n, \Delta t_r) = \Delta t_r$	undefined	$\delta_{int}(n, \Delta t_r) = (n+1, \Delta t_r{}')$ $\Delta t_r{}'$ r.s.f. Exponential($\frac{1}{480}$)

$X = \{\}$ and $Y = \mathbb{N}$ (Counting G. & Random G.); $X = Y = \mathbb{N}$ (Simple P. & Loyal P. & Ideal P.)
$S = \mathbb{N}$ (Counting G.); $S = \mathbb{N} \cup \{\varnothing\}$ (Simple P. & Ideal P.); $S = (\mathbb{N} \cup \{\varnothing\}) \times (\mathbb{R}_0^+ \cup \{\infty\})$ (Loyal P.); $S = \mathbb{N} \times \mathbb{R}^+$ (Random G.)
$\lambda(n) = n$ (Counting G. & Simple P. & Ideal P.); $\lambda(n, \Delta t_r) = n$ (Loyal P. & Random G.)

As given by (3), the optimal time resolution of a coupled model depends on its components. The resolution is 0 if any component has an optimal resolution of 0. Otherwise it is the greatest common divisor of the component resolutions.

$$\mathrm{tr}(M) = \sup_{\Delta t \in \mathbb{Q}^+} (\forall d \in D, (\mathrm{tr}(M_d) > 0) \wedge (\Delta t | \mathrm{tr}(M_d))) \quad (3)$$

With the theory established, we consider a more practical approach for selecting time resolutions in a simulation tool with integer-based time values. In this context, the time resolution assigned to a model must be sufficiently fine, but it need not be optimal. It is reasonable to allow simulation software users to choose time resolutions from a geometric sequence of predefined values. Below is one such sequence.

10^6 years, 10^3 years, years, days, hours, minutes, seconds,
10^{-3} seconds, 10^{-6} seconds, ..., 10^{-36} seconds

Let us revisit the models in Table 7. The Counting Generator has an optimal time resolution of 8 minutes, so from the list a user would choose "minutes". The optimal resolution of both Simple and Loyal processors is 15 seconds, so "seconds" is the practical choice. For models with infinite optimal time resolutions, like the Ideal Processor, a "N/A" option should be available. Users must exercise their judgment for models with optimal time resolutions of 0. The Random Generator has a mean interval of 8 minutes, so a 1-minute resolution is perhaps too coarse, a 1-microsecond resolution may be overly conservative, but seconds or milliseconds seem reasonable.

Since every resolution in the sequence is divisible by its successor, a coupled model can be assigned the finest time resolution of all of its components. The time resolution of a simulation run may be no coarser than that of the model or that of the time values in the input series.

5. ENCAPSULATION

A model's interface includes all information that may need to be referenced by a simulation run which depends on the model, or by a coupled model that includes it as a component. Information that is not part of the interface may be considered encapsulated. In the original conventions, the interface consists of only the input and output sets X and Y, so the state of an atomic model and the composition of a coupled model are both encapsulated.

The challenge is how to ensure model states and compositions remain encapsulated as the task shifts from specifying models to running simulations. Directly or indirectly, simulation runs must be configured with initial model states. Also, there is often a need to report on statistical information residing in a model's final state. If an atomic model's initial and final states are exposed in the simulation configuration and results, as they are in the specification-oriented conventions in Table 5, the model's state can not be considered encapsulated. If the initial and final states of the components of a coupled model are exposed, its composition is not encapsulated.

Our solution begins with the inclusion of a list of parameters in every model, as well as a corresponding list of parameter values in every simulation run. In an atomic model, the parameters are used to derive the initial state, allowing the state variables to be hidden from the encompassing simulation or coupled model. This is common practice, as 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. Our implementation-oriented atomic model conventions therefore include a list of constants, parameter-dependent variables that never change once a simulation is underway. We also propose that atomic models have two initialization functions. The first to be invoked is the constant initialization function, the only function in which parameters can be accessed and constants can be modified. The ability to modify constants in this function is important for populating arrays and data structures that will later be immutable. The second initialization function is the state initialization function, kept separate from the first so that modifications to the constants can be disallowed before the state variables acquire their initial values.

As mentioned above, our conventions require every model to include a parameter list, as well as an initialization function that reads parameter values at the beginning of a simulation. We propose that every model also include a statistic list, as well as a finalization function that produces statistic values at the end of a simulation. In an atomic model, the finalization function extracts information from the state variables without exposing them, thereby keeping the state of the model encapsulated. It also provides a place to release any computer resources acquired during initialization. We recommend that state changes be permitted in the finalization function, and that the elapsed time be made available to inform these changes. This makes it convenient to account for any continuous change in a system over the time period between the last transition time and the end time of a simulation run.

A simple way to derive the parameter and statistic lists of a coupled model is to concatenate the parameter and statistic lists of its components. Unfortunately, a simulation tool that does this automatically would expose information about a coupled model's composition. It would also furnish a coupled model with redundant parameters in cases where multiple components require the same parameter value. Our conventions require a coupled model's parameters and statistics to be listed independently of those of its components. At the beginning of a simulation run, a coupled model's initialization function reads its parameter values and prepares a separate list of parameter values for each component. Afterwards, the initialization function of each component is invoked with the appropriate list. At the end of a simulation run, the finalization function of each component produces a separate list of statistic values. Afterwards, the coupled model's finalization function is invoked to read each list 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 latest component-level event, the time elapsed since the beginning of the simulation, and the time remaining until the next component-level internal transition.

6. CONCLUSION

Convenience, efficiency, repeatability, and other practical considerations motivate the implementation-oriented DEVS conventions proposed in this paper. Although our conventions differ in appearance from the original mathematical conventions, they remain similar in principle by keeping a model's state or composition encapsulated and by allowing fine control over the timing of events. Depending on the specific requirements of future simulation tools, the proposed conventions may be adopted in part or in their entirety. In either case, the developers of these tools will begin their projects with a greater awareness of the technological issues they will inevitably face, and at least one practical solution for many of these issues. We encourage others to propose implementation-oriented conventions for Parallel DEVS, Real Time DEVS, and other DEVS variants.

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.

Chow, A. C. H. and B. P. Zeigler (1994). Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism. In *Proceedings of the Winter Simulation Conference*, pp. 716–722.

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*, pp. 1518–1523.

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*. Boca Raton, FL, USA: CRC Press.

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.