# A NEW MODELING INTERFACE FOR SIMULATORS IMPLEMENTING THE DISCRETE EVENT SYSTEM SPECIFICATION

James Nutaro

Computational Sciences and Engineering Division
Oak Ridge National Laboratory
One Bethel Valley Road
Oak Ridge, TN, USA
nutarojj@ornl.com

## ABSTRACT

The Discrete Event System Specification (DEVS) offers a unique modeling interface that is often perplexing to modelers more familiar with other simulation paradigms. Recent advances in the use of super dense time for discrete event simulation offer an opportunity to recast the traditional interface into a form less confounding for new users. The new interface proposed here allows a natural progression from a message oriented approach to modeling to the familiar DEVS approach. The proposed approach retains the expressive power of the DEVS formalism, and in this sense represents a simple repackaging of the DEVS approach into a more intuitively appealing form.

**Keywords:** agent based model, DEVS, discrete event simulation

## 1 INTRODUCTION

Simulation tools based on the Discrete Event System Specification (DEVS) offer a particular modeling interface to the user. To build a basic model component the user defines five functions: the time advance, output, and internal, external, and confluent state transition functions. Large models are built by connecting components and assemblages of components. Outputs from components are transmitted indirectly via a coupling graph provided by the modeler as part of the definition of each assemblage.

In part, this modeling interface is chosen because it reflects the canonical definition of a DEVS model (Zeigler, Muzy, and Kofman 2018). For modelers aware of the foundations of DEVS, its motivations, and its capabilities this modeling interface is entirely satisfactory. On the other hand, if the modeler's prior experience is with the classical world views (see, e.g., Carson 1993) or the logical processes view prevalent in parallel discrete event simulation (PDES) (see, e.g., Fujimoto 2000) then this interface can be perplexing.

It can be argued that this confusion is due to the simulation programmer intermingling the model specification and its implementation (i.e. programming). This phenomenon is not limited to discrete event simulations. The same confusion can be found in simulations of continuous systems where the numerical integration method and the model equations are tightly intertwined in the simulation program. An effective remedy is to use tools that enforce separation. Examples include the Modelica language for (mostly) continuous models (Fritzson 2011) and the DEVS-CML language for DEVS models (Hollmann, Cristiá, and Frydman 2015, Cristiá, Hollmann, and Frydman 2019), to name just two. These permit the modeler to think in terms of suitably abstract, but precise, modeling constructs which the compiler transforms into a general purpose programming language.

Nonetheless, general purpose programming languages continue to be popular for building simulation models, but the software libraries that facilitate this do not enforce the intended use of modeling constructs. Instead, they rely on familiarity of the modeler with the proper use of these constructs or that their proper use be intuitively appealing. Our goal here is to offer a modeling interface the retains much of the expressive power of DEVS and whose proper use is intuitively appealing.

Towards this goal, relatively recent developments concerning super dense time in discrete event models have created an opportunity to reconsider the traditional DEVS modeling interface. In particular, by using super dense time the canonical DEVS simulation algorithm can be reformulated to eliminate the distinct phase of gathering output. This produces a simulation procedure that is much closer in form to the classical world views and does not require the explicit definition of time advance and output functions.

We take advantage of this revised simulation procedure to present a new interface that permits a natural progression from a message oriented approach to modeling, similar in most respects to what is offered by parallel discrete event simulation tools, to the familiar DEVS interface. At each step the proposed approach preserves the natural handling of simultaneity that is one of the strengths of the DEVS framework.

## 2 BASIC MODELS

In common with DEVS and PDES simulation tools, the proposed modeling interface centers around the concept of a basic model. This basic model is analogous to a DEVS atomic model and PDES logical process. The basic model has a state accessible only to it, and basic models exchange information solely by passing messages to one another. At each point in simulation time, a model has exactly one state, and any future state is strictly a function of states and messages in the past. This latter requirement is satisfied while still allowing for instantaneous events by using super dense time (see, e.g., Nutaro 2010, Lee and Zheng 2005, Sarjoughian and Sundaramoorthi 2015, Maler, Manna, and Pnueli 1992, Mosterman, Simko, Zander, and Han 2014).

### 2.1 Super dense time

The form of super dense time we will use here comprises pairs $(a,b)$ in which $a$ is a physically meaningful measure of time in the context of the model and $b$ arranges activities occurring at the same $a$. The ordering of time pairs is lexicographical. Two operators act on the super dense time set. These are an advance operator $+$ and interval length $\ell$ such that

$$(a,b)+(c,d)=\begin{cases}(a,b+d) & \text{if } c=0\\ (a+c,d) & \text{if } c\neq 0\end{cases} \text{ and}$$

$$\ell[(a,b),(c,d))=\begin{cases}(c-a,d) & \text{if } a\neq c\\ (0,d-b) & \text{if } a=c\end{cases}.$$

Because $\ell$ operates on an interval $[(a,b),(c,d))$ it is only defined when $(a,b)\leq(c,d)$.

For brevity we will often refer to a time instant $(a,b)$ as $t$. The value 0 can substitute for the pair $(0,0)$ and 1 for the pair $(0,1)$. The former is justified by $(0,0)+t=t+(0,0)=t$, and the latter by $(0,1)$ being the smallest $t>0$. A single $\infty$ will stand for the pair $(\infty,\infty)$. Observe that $+$ does not commute but it is associative. This choice of super dense time is somewhat arbitrary as there are alternatives that could be used without changing the modeling interface constructed below.

## 2.2 The modeling interface

Each basic model has a state $q$. The proposed modeling interface ensures that $q$ is a function of time so the notation $q(t)$ refers to the unique state at time $t$. The state evolves via two methods provided by the modeler.

1. The method *initialize* assigns an initial state to the model at the starting time $t_0$. This method returns the first instant $t > t_0$ at which $q(t_0) \neq q(t)$; that is, the first instant at which a new state should be assigned to the component.
2. The method *update* is called at the time $t$ returned by *initialize* or the prior call to *update*, or at some earlier time $t$ if a message is sent to the component. This method assigns a new state to the model at time $t$. For this purpose, the modeler is provided with the current time and all messages arriving at that time. As with *initialize* it returns the first instant $t' > t$ such that $q(t') \neq q(t)$.

Basic models exchange messages by using a pair of methods.

1. The method *send* transmits a message to a model. This method is provided by the simulator to the model. The method arguments are the model sending the message, the model to receive the message, and the data to be transmitted. All messages sent at time $t$ are delivered in a single call to *update* at the next instant of time $t+1$.
2. The method *relay* allows a model that is sent a message to pass it along to another component as though the message had been sent by the relaying component at the current time. As with the *send* message, a message that is relayed at time $t$ is delivered with all other messages at the next instant $t+1$. This method is provided by the modeler.

This interface requires simulation time to strictly advance, with the smallest possible advance in time being equal to 1. The state changes at discrete instants, and each change is marked by a single call to *initialize* or *update*.

Unlike many PDES tools, messages are not assigned time stamps by the modeler. Rather, in a fashion similar to DEVS, sending a message reschedules the target to undergo a change of state at the next instant of simulation time. This is consistent with many modeling approaches - such as cellular automata, difference equations, and DEVS - in which the state $q(t+1)$ is a function of $q(t)$ and output produced (i.e., messages sent) at $t$. Unlike a typical DEVS modeling interface, the proposed approach removes the distinct output, time advance, and state transition methods, which are often unfamiliar and confusing to modelers coming from other simulation paradigms.

## 2.3 Example #1

Consider a model of a server and queue. New jobs are processed first in, first out. When a job is complete, the model reports which job was finished and how many jobs remain in the queue. The state of each model is a queue of pending jobs each with its own time to complete and the time that has elapsed since the prior change of state. A DEVS realization of this model might be as follows.

**External state transition function**: If the queue is not empty, then reduce the time to complete the first job in the queue by the elapsed time. Place new jobs at the back of the queue.
**Internal state transition function**: Remove the job at the front of the queue.

**Confluent state transition function**: Remove the job at the front of the queue. Place new jobs at the back of the queue.
**Time advance**: If the queue is empty then return $\infty$ otherwise return the time to completion of the first job in the queue.
**Output function**: Return the job at the front of the queue and the queue length minus one.

When viewed through the lens of general systems theory (Nutaro 2010, Zeigler, Muzy, and Kofman 2018, Mesarovic and Takahara 1989, Klir 1969), this implementation neatly separates the model's distinct aspects: the production of output based on the current state $q(t)$, the computation of the new state $q(t+1)$ from the current state $q(t)$ and input at $t$, and the time advance (lifetime) of the state. Hence its appeal to practitioners whose initial exposure to simulation was through the general systems paradigm.

However, this realization raises questions when approached from the perspective of a simulation programmer. For instance, with many DEVS based tools we could remove the job at the front of the queue as part of the output function without changing the behavior of the simulation program. So why do we access the front of the queue twice: once in the output function and again in the internal and confluent transition functions?

Similarly, why do we return the queue length minus one in the output and then actually shrink the queue size in the state transition function? Indeed, we might argue that the following implementations of the internal and confluent transition functions and the output function are simpler and therefore more desirable.

**Internal state transition function**: Do nothing.
**Confluent state transition function**: Place new jobs at the back of the queue.
**Output function**: Remove the job at the front of the queue and return it with the queue size.

This eliminates the internal transition function and removes a superfluous subtraction. The new code, it could be argued, is shorter, simpler, and faster because it has one fewer operation - the unnecessary subtraction.

The potential appeal of the proposed modeling interface is made apparent by this model. In a new implementation, we retain the queue of the DEVS model but replace the elapsed time with the time of last event $t_L$. The letter $h$ will indicate the time remaining to finish the job at the front of the queue and now means the current simulation time. The *initialize* and *update* methods are as follows.

**Initialize**: Set $t_L$ to now. Return $\infty$.
**Update**: If the queue is not empty, then set $h$ to $\ell[\text{now}, t_L + h)$; if $h$ is zero then remove that job from the queue and *send* it with the queue size. Insert each arriving job at the back of the queue. Set $t_L$ to now. If the queue is not empty, return now plus the completion time of the job at the front of the queue. Otherwise return $\infty$.

This model performs the same actions as the DEVS model, and the input and output trajectories of the two models will be identical. However, the simulation programmer may prefer this realization for the same reasons that the internal transition function was removed from the DEVS implementation: it eliminates apparent redundancy.

## 2.4 Example #2

The types of questions illustrated in the first example emerged early in the process of modeling a manufacturing facility (Nutaro, Schryver, and Haire 2012). In this application, holding tanks are modeled as discrete event systems. The input to the tank is its net fill rate, which is positive if the tank is filling and negative if the tank is draining. The net fill rate is the sum of input arriving from upstream chemical processes and extraction by downstream chemical processes.

The output of the tank model can be full, which causes the upstream processes to stop; empty, which causes the downstream processes to stop; or available, which allows all processes to operate. The state of the tank comprises its net fill rate, fluid level, time $\sigma$ remaining until the next event, the output $y$ to be produced at that time, and the time elapsed since the prior change of state. A motivating factor in the design of its DEVS realization was to avoid spurious output. Its DEVS model, somewhat simplified, is as follows.

**External state transition function**: Update the fluid level using the current net fill rate and elapsed time. Store the new net fill rate. There are two possibilities for the next event.

1 If the tank is neither full nor empty, then set $\sigma$ to the time remaining until one of these events will happen and set $y$ accordingly; whether this is full or empty depends on the sign of the net fill rate. If this is zero then set $\sigma$ to infinity.

2 If the tank is empty and the net fill rate is positive, or if the tank is full and the net fill rate is negative, then set $\sigma$ to zero and $y$ to available. Otherwise set the net fill rate to zero and $\sigma$ to infinity.

**Internal state transition function**: If $\sigma = 0$ then we just signaled available. Set $\sigma$ to the time until the tank becomes empty or full and set $y$ accordingly. Otherwise, update the fluid level using the net fill rate and $\sigma$. This new fluid level will be full or empty. Set $\sigma$ to infinity.

**Confluent state transition function**: Invoke the internal transition function followed by the external transition function.

**Time advance**: Return $\sigma$.

**Output function**: Return $y$.

This model has two conceptually challenging elements. First is the zero time event scheduled by the external transition function. This is necessary to signal the change of condition from full or empty to available, but it requires remembering its occurrence in the internal transition function, which is invoked in the following instant of simulation time. Second is the apparently spurious internal transition function. As in the prior example, its actions could be moved into the output function.

The potential appeal of the proposed modeling interface is that it dispenses with these conceptual challenges. In particular, a change of state in the condition of the tank is signalled directly from the *update* method without requiring the use of a zero time event. A new realization of the tank model might be as follows. Again, we retain the state variables of the DEVS model but replace the elapsed time with the time of last event $t_L$.

**Initialize**: Set $t_L$ to now. Return $\infty$.

**Update**: Update the fluid level using the net fill rate and real part of now minus $t_L$. Store the new net fill rate. There are two possibilities for the output.

1 If the tank is empty and the net fill rate is positive, or if the tank is full and the net fill rate is negative, then *send* available.

2 If the tank is empty and the net fill rate is not positive, or if the tank is full and the net fill rate is not negative, then *send* full or empty as appropriate and set the net fill rate to zero.

Set $t_L$ to now. Return now plus the time remaining to fill or empty the tank, or infinity if the net fill rate is zero.

## 2.5 Simulating an atomic DEVS

The notation used in this section assumes the reader is familiar with DEVS; if not, then see, e.g., Zeigler, Muzy, and Kofman (2018). The proposed modeling approach produces event trajectories. Hence, a DEVS atomic model can always be built that exhibits the same behavior as a model built with the new interface (Zeigler, Muzy, and Kofman 2018). In the other direction, we may simulate any DEVS atomic model using the proposed interface. To see how, let the initial state of the DEVS be $q$ with elapsed time $0 \leq e \leq ta(q)$. The simulator retains the time of last event $t_L$ for the purposes of updating the elapsed time $e$ at each event.

The *initialize* method for this atomic model simulator is listed as algorithm 1. If the elapsed time is equal to the time advance, then $t$ is the last instant at which the current state holds. The algorithm sends $\lambda(q(t))$ and schedules the new state to be assigned at $t+1$. Otherwise, the time advance is positive and the algorithm schedules output to occur when the time advance expires.

```
**Input:** Current time $t$
**Output:** Time of next event
$t_L \leftarrow t$;
**if** $e = ta(q)$ **then**
    send($\lambda(q)$);
    **return** $t+1$;
**else**
    **return** $t + \ell[e, ta(q))$;
**end**
```

**Algorithm 1:** The *initialize* method.

The *update* method is listed as algorithm 2. It begins by updating the elapsed time and then the time of last event. If the elapsed time is equal to the time advance and no input has arrived, then $t$ is the last instant at which $q(t_L)$ holds. However, $q$ has not expired so $q(t) = q(t_L)$. Hence, we send the output $\lambda(q(t))$ and schedule a change of state in the next instant.

If $e$ is less than or equal to the time advance and input is available, then the state $q$ has not expired but input was generated in the previous simulation instant that should be used to calculate $q(t) \neq q(t_L)$. In this case, we calculate $q(t)$ using the external transition function. The time passed to $\delta_{ext}$ is the elapsed time at the moment the incoming event was generated, which is the instant prior to $t$.

The only other possibility is that $e = ta(q) + 1$. This case occurs if an output was generated in the prior instant of simulation time. If input was also sent to the model at that prior instant, then the new state $q(t)$ is calculated using the confluent transition function. Otherwise the new state is given by the internal transition function.

After calculating the model state, the *update* method then behaves just like the *initialize* method. If the elapsed time is equal to the time advance, we generate output $\lambda(q(t))$ and schedule a change of state for $t+1$. Otherwise, schedule an output to occur $ta(q(t))$ units of time into the future.

To illustrate the operation of these algorithms, consider the server with queue described in Section 2.3. For the first example, let the queue be empty and $e = 0$. The first job arrives at time $(1,0)$ and needs $(0,1)$ units of time to complete. The second job arrives at time $(1,10)$ and needs $(1,0)$ units of time to complete.

The state trajectory for this model as it would be calculated by a DEVS simulator is shown in Table 1. The method of calculation using super dense time follows naturally from the familiar simulation procedure by accounting explicitly for the distinct, but sequentially occurring, activities of output production followed by the change of state (Nutaro 2010).

```
**Input:** Current time $t$ and input $x$
**Output:** Time of next event
$e \leftarrow e + \ell[t_L, t)$;
$t_L \leftarrow t$;
if $e = ta(q)$ and $x$ is empty then
    send($\lambda(q)$);
    return $t + 1$;
else if $e \leq ta(q)$ and $x$ is not empty then
    $(a, b) \leftarrow e$;
    $e \leftarrow 0$;
    $q \leftarrow \delta_{ext}(q, (a, b - 1), x)$;
else if $e = ta(q) + 1$ and $x$ is not empty then
    $e \leftarrow 0$;
    $q \leftarrow \delta_{con}(q, x)$;
else
    $e \leftarrow 0$;
    $q \leftarrow \delta_{int}(q)$;
end
if $ta(q) = 0$ then
    send ($\lambda(q)$);
    return $t + 1$
else
    return $t + ta(q)$;
end
```

**Algorithm 2:** The *update* method.

The sequence of *initialize* and *update* calls that effect the simulation of this model are listed below. Recall that input $x$ appearing at time $t$ in Table 1 implies a *send* call at time $t$ so that the message affects the new state at $t+1$. This new state is calculated with the *update* method.

1. *initialize*$(t = 0)$. Set $t_L \leftarrow 0$. Returns $0 + \ell[0, \infty) = \infty$.
2. *update*$(t = (1,1), x = \{(0,1)\})$. Set $e \leftarrow 0 + \ell[0, (1,1)) = (1,1)$ and $t_L \leftarrow (1,1)$. Because $e < ta(q)$ and $x$ is not empty, the external transition function is called with $x$ and elapsed time argument $(1,0)$, which corresponds to when $x$ was generated. The new state has $(0,1)$ at the front of the queue and the time advance is now $(0,1) > 0$. Set $e \leftarrow 0$ and return $t + (0,1) = (1,2)$.
3. *update*$(t = (1,2), x = \phi)$. Set $e \leftarrow 0 + \ell[(1,1), (1,2)) = (0,1)$ and $t_L \leftarrow (1,2)$. Because $e = ta(q)$ and $x$ is empty, the output 0 is sent and the call returns $(1,3)$.
4. *update*$(t = (1,3), x = \phi)$. Set $e \leftarrow (0,1) + \ell[(1,2), (1,3)) = (0,2)$ and $t_L \leftarrow (1,3)$. Because $x$ is empty and $e = ta(q) + 1$, the new state is calculated with the internal transition function. This results in an empty queue and infinity time advance. Set $e \leftarrow 0$ and return $\infty$.
5. *update*$(t = (1,11), x = \{(1,0)\})$. Set $e \leftarrow 0 + \ell[(1,3), (1,11)) = (0,8)$ and $t_L \leftarrow (1,11)$. The elapsed time is less than the time advance and $x$ is not empty so the new state is determined by the external

Table 1: A simulation trajectory for the server with queue.

| $t$ | queue | $e$ | ta($q$) | $x$ | $\lambda(q)$ |
|---|---|---|---|---|---|
| $(0,0)$ | $\phi$ | $(0,0)$ | $\infty$ | $\phi$ | $\phi$ |
| $(1,0)$ | $\phi$ | $(1,0)$ | $\infty$ | $(0,1)$ | $\phi$ |
| $(1,1)$ | $(0,1)$ | $(0,0)$ | $(0,1)$ | $\phi$ | $\phi$ |
| $(1,2)$ | $(0,1)$ | $(0,1)$ | $(0,1)$ | $\phi$ | $0$ |
| $(1,3)$ | $\phi$ | $(0,0)$ | $\infty$ | $\phi$ | $\phi$ |
| $(1,10)$ | $\phi$ | $(0,7)$ | $\infty$ | $(1,0)$ | $\phi$ |
| $(1,11)$ | $(1,0)$ | $(0,0)$ | $(1,0)$ | $\phi$ | $\phi$ |
| $(2,0)$ | $(1,0)$ | $(1,0)$ | $(1,0)$ | $\phi$ | $0$ |
| $(2,1)$ | $\phi$ | $(0,0)$ | $\infty$ | $\phi$ | $\phi$ |

Table 2: A second example of a simulation trajectory for the server with queue.

| $t$ | queue | $e$ | ta($q$) | $x$ | $\lambda(q)$ |
|---|---|---|---|---|---|
| $(0,0)$ | $(2,0)$ | $(1,0)$ | $(2,0)$ | $\phi$ | $\phi$ |
| $(1,0)$ | $(2,0)$ | $(2,0)$ | $(2,0)$ | $(0,0)$ | $0$ |
| $(1,1)$ | $(0,0)$ | $(0,0)$ | $(0,0)$ | $\phi$ | $0$ |
| $(1,2)$ | $\phi$ | $(0,0)$ | $\infty$ | $\phi$ | $\phi$ |

transition function. The elapsed time argument passed to $\delta_{ext}$ is $(1,10)$ corresponding to when $x$ was generated. The elapsed time is set to zero, the time advance is $(1,0)$ and we return $t+(1,0)=(2,0)$.

6. *update*$(t=(2,0), x=\phi)$. Set $e \leftarrow 0+\ell[(1,11),(2,0))=(1,0)$ and $t_L \leftarrow (2,0)$. Because $e$ is equal to the time advance and $x$ is empty, we send the output 0, and return $(2,1)$.
7. *update*$(t=(2,1), x=\phi)$. Set $e \leftarrow (1,0)+\ell[(2,0),(2,1))=(1,1)$ and $t_L \leftarrow (2,1)$. Because $e = ta(q)+1$ and $x$ is empty, the next state is calculated with the internal transition function. The queue becomes empty, $e$ is set to zero, and we return $\infty$.

For a second example, let the queue contain a single job with time remaining of $(2,0)$, elapsed time of $(1,0)$, and new input with processing time 0 arriving at $(1,0)$. The initial simulation time is zero. The simulation trajectory for this model is shown in Table 2. The sequence of *initialize* and *update* calls that effect the simulation of this model are listed below. Recall that the message set at time $(1,0)$ will be delivered via *update* at $(1,1)$ to be used when calculating $q((1,1))$.

1. *initialize*$(t=0)$. Set $t_L \leftarrow 0$ and return $0+\ell[(1,0),(2,0))=(1,0)$.
2. *update*$(t=(1,0), x=\phi)$. Set $e \leftarrow (1,0)+\ell[0,(1,0))=(2,0)$ and $t_L \leftarrow (1,0)$. Because $e=ta(q)$ and $x$ is empty, send the output 0 and return $t+1=(1,1)$.
3. *update*$(t=(1,1), x=\{(0,0)\})$. Set $e \leftarrow (2,0)+\ell[(1,0),(1,1))=(2,1)$ and $t_L \leftarrow (1,1)$. Because $e=ta(q)+1$ and $x$ is not empty, calculate the new state using the confluent transition function. This removes the first job from the queue and adds the second job. The elapsed time $e$ is set to zero. The new time advance is $(0,0)$. We send the output 0 and return $t+1=(1,2)$.
4. *update*$(t=(1,2), x=\phi)$. Set $e \leftarrow 0+\ell[(1,1),(1,2))=(0,1)$ and $t_L \leftarrow (1,2)$. Because $x$ is empty and $e=ta(q)+1$, the new state is calculated with the internal transition function. This results in an empty queue and infinite time advance. Set $e \leftarrow 0$ and return $\infty$.

## 3 HIERARCHICAL MODELS

Direct communication between model components via the *send* method is an intuitively appealing way to combine components into a larger model. However, this approach leads to a model structure in which components cannot be easily disentangled for reuse, component level testing, and a variety of other purposes (e.g., as described in Varga and Hornig 2008, Nutaro 2010, Zeigler, Muzy, and Kofman 2018, Sarjoughian and Elamvazhuthi 2009). To address this shortcoming, the *relay* method is included in the modeling interface. By using this method, hierarchical models can be simulated using a recursive routing procedure similar to the one described by Muzy and Nutaro (2005).

Recall that the *send* method is provided by the simulation engine. When a model invokes *send*, this causes the simulator to invoke the *relay* method of the destination. The *relay* method returns a set of models that are to receive the message. If this set is empty then the message is discarded. If this set contains the destination itself, then the new message is added to the messages that will be delivered to it in the next simulation instant. For every other component in the returned set, the simulator uses *send* to deliver that component a message.

To route messages in a hierarchical model, components and assemblages are arranged into a tree in the same manner that coupled models are constructed with DEVS (Zeigler, Muzy, and Kofman 2018). The *relay* message of each basic component returns a set containing that component alone. Each assemblage uses the *relay* method to realize its coupling graph.

Specifically, if *relay* is provided with a message originating from a child of the assemblage, then this is routed according to the internal coupling of the model. The returned set may include children of the assemblage and its parent if the message is to be an output from the assemblage itself. Otherwise, if *relay* is provided with a message originating from its parent, then this is input to the network and the returned set may include the assemblage's children.

This hierarchical routing procedure is summarized as algorithm 3. The listing assumes an object oriented implementation in which *self* refers to the object whose method has been invoked.

```
Function relay(message,originator):
    if self is a basic component then
        $R \leftarrow \{self\}$;
    else if originator = self.parent then
        $R \leftarrow$ the children to receive input to the assemblage;
    else
        $R \leftarrow$ the children to receive the message and, if this is an output of the network, self.parent ;
return $R$
Function send(source,message,target):
    $R \leftarrow$ target.relay(message,source);
    foreach model $\in R$ do
        if model = target then
            Deliver message to target at $t+1$;
        else
            send(target,message,model)
        end
    end
return
```

**Algorithm 3:** Hierarchical routing using the *send* and *relay* methods.

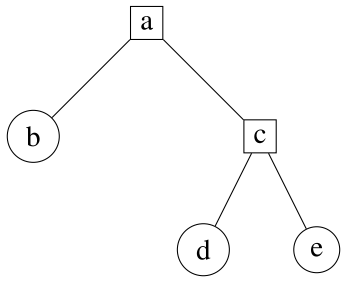

Figure 1: A model tree to illustrate the execution of algorithm 3.

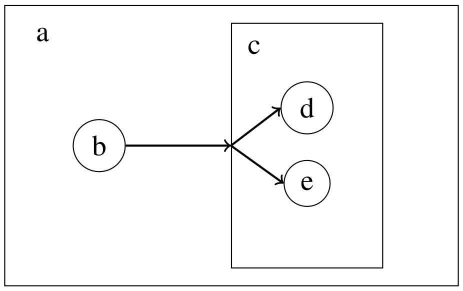

Figure 2: Coupling graphs for each level of the model shown in Figure 1.

To see how the algorithm operates, consider the model tree in Figure 1. Its coupling graphs at each level are shown in Figure 2. Components $b$ and $c$ are siblings, with $b$ connected to $c$. Input to $c$ is delivered to its children $d$ and $e$. The component $b$ initiates the routing algorithm by sending a message $x$ to $c$ with the call $\text{send}(b,x,a)$. This results in the call $a.\text{relay}(x,b)$, and the coupling graph indicates the return set $R = \{c\}$. Because $a \neq c$, the algorithm calls $\text{send}(a,x,c)$.

This recursive call to send causes a call to $c.\text{relay}(x,a)$. The coupling graph indicates that the return set is $R = \{d,e\}$. Because $d \neq c$ and $e \neq c$, calls are made to $\text{send}(c,x,d)$ and $\text{send}(c,x,e)$. We will follow the call to $\text{send}(c,x,d)$. This causes a call to $d.\text{relay}(x,c)$. Because $d$ is a basic component, the returned set is $R = \{d\}$ and since $d = d$ we add $x$ to the messages that will be delivered to it in the next instant. After the call to $\text{send}(c,x,e)$ is resolved in the same fashion, all of the send calls return and the routing algorithm terminates.

## 4 CONCLUSION

The proposed modeling interface offers a way for users new to DEVS to take advantage of its attractive features while retaining an intuitive approach to simulation programming. At its most basic level, the modeler describes behavior using an update method and a message passing mechanism similar to what is offered by parallel discrete event simulation tools and packages like OMNeT++ (Varga and Hornig 2008). This can evolve naturally to hierarchical modeling and the use of a typical DEVS interface. An implementation of the proposed modeling interface in C++ is available online (Nutaro 2018).

A key concept in the new approach is that an event scheduling algorithm operating on super dense time can replicate the two fundamental steps of a DEVS simulation algorithm. These steps are (1) calculating output for all models imminent at the next event time and (2) calculating the states for all active components using these outputs. This two step procedure avoids serializing the execution of events at a given simulation time, and is the basis for being able to correctly and naturally simulate discrete time models within the DEVS framework. The utility of this aspect of DEVS is apparent in many applications, particularly those inspired by cellular automata (see, e.g., Wainer 2009).

A potentially important outcome of the reformulated modeling interface is accelerated growth in the use of DEVS as an agent based modeling tool. The usefulness of DEVS as an agent based modeling formalism has been widely demonstrated (see, e.g., Seck, Frydman, and Giambiasi 2005, Akplogan, Quesnel, Garcia, Joannon, and Martin-Clouaire 2010, Bisgambiglia, Bisgambiglia, and Franceschini 2013 and the broader discussion by Uhrmacher and Weyns 2009). That a discrete event modeling formalism can be readily used for agent based modeling is particularly noteworthy given the perceived divide between agent based and discrete event simulation (Siebers, Macal, Garnett, Buxton, and Pidd 2010). Nonetheless, this development appears to be largely unnoticed by modelers working outside of the DEVS paradigm. By offering an intuitively appealing modeling interface in a DEVS simulation tool we take a step towards bridging this gap.

## ACKNOWLEDGEMENTS

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

## REFERENCES

Akplogan, M., G. Quesnel, F. Garcia, A. Joannon, and R. Martin-Clouaire. 2010. “Towards a Deliberative Agent System Based on DEVS Formalism for Application in Agriculture”. In *Proceedings of the 2010 Summer Computer Simulation Conference*, SCSC ’10, pp. 250–257. San Diego, CA, USA, Society for Computer Simulation International.

Bisgambiglia, P.-A., P. A. Bisgambiglia, and R. Franceschini. 2013. “Agent-oriented Approach Based on Discrete Event Systems (WIP)”. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, DEVS 13, pp. 27:1–27:6. San Diego, CA, USA, Society for Computer Simulation International.

Carson, J. S. 1993. “Modeling and Simulation Worldviews”. In *Proceedings of 1993 Winter Simulation Conference - (WSC ’93)*, pp. 18–23.

Cristiá, M., D. A. Hollmann, and C. Frydman. 2019. “A multi-target compiler for CML-DEVS”. *SIMULATION* vol. 95 (1), pp. 11–29.

Fritzson, P. 2011. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Wiley-IEEE Press.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. Wiley.

Hollmann, D. A., M. Cristiá, and C. Frydman. 2015. “CML-DEVS: A specification language for DEVS conceptual models”. *Simulation Modelling Practice and Theory* vol. 57, pp. 100 – 117.

Klir, G. J. 1969. *An approach to general systems theory*. Van Nostrand Reinhold Co.

Lee, E. A., and H. Zheng. 2005. “Operational Semantics of Hybrid Systems”. In *Hybrid Systems: Computation and Control*, edited by M. Morari and L. Thiele, pp. 25–53. Berlin, Heidelberg, Springer Berlin Heidelberg.

Maler, O., Z. Manna, and A. Pnueli. 1992. “From timed to hybrid systems”. In *Real-Time: Theory in Practice*, edited by J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, pp. 447–484. Berlin, Heidelberg, Springer Berlin Heidelberg.

Mesarovic, M. D., and Y. Takahara. 1989. *Abstract Systems Theory*. Springer-Verlag.

Mosterman, P. J., G. Simko, J. Zander, and Z. Han. 2014. “A Hyperdense Semantic Domain for Hybrid Dynamic Systems to Model Different Classes of Discontinuities”. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC ’14, pp. 83–92. New York, NY, USA, ACM.

Muzy, A., and J. J. Nutaro. 2005. “Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators”. In *1st Open International Conference on Modeling & Simulation*, pp. 273–279.

James Nutaro 2018. “A discrete event system simulator”. https://sourceforge.net/projects/bdevs/.

Nutaro, J. J. 2010. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley.

Nutaro, J. J., J. C. Schryver, and M. J. Haire. 2012. “The throughput, reliability, availability, maintainability (TRAM) methodology for predicting chemical plant production”. *2012 Proceedings Annual Reliability and Maintainability Symposium*, pp. 1–6.

Sarjoughian, H. S., and V. Elamvazhuthi. 2009. “CoSMoS: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation”. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools ’09, pp. 59:1–59:9. ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. “Superdense Time Trajectories for DEVS Simulation Models”. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS ’15, pp. 249–256. San Diego, CA, USA, Society for Computer Simulation International.

Seck, M., C. Frydman, and N. Giambiasi. 2005. “Using DEVS for Modeling and Simulation of Human Behaviour”. In *Artificial Intelligence and Simulation*, edited by T. G. Kim, pp. 692–698. Berlin, Heidelberg, Springer Berlin Heidelberg.

Siebers, P. O., C. M. Macal, J. Garnett, D. Buxton, and M. Pidd. 2010, Sep. “Discrete-event simulation is dead, long live agent-based simulation!”. *Journal of Simulation* vol. 4 (3), pp. 204–210.

Uhrmacher, A. M., and D. Weyns. (Eds.) 2009. *Multi-Agent Systems: Simulation and Applications*. CRC Press.

Varga, A., and R. Hornig. 2008. “An Overview of the OMNeT++ Simulation Environment”. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools ’08, pp. 60:1–60:10. ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation: A Practitioner’s Approach*. CRC Press.

Zeigler, B., A. Muzy, and E. Kofman. 2018. *Theory of Modeling and Simulation, 3rd Edition*. Academic Press.

## AUTHOR BIOGRAPHIES

**JAMES NUTARO** is a member of the research staff at Oak Ridge National Laboratory in Oak Ridge, Tennessee and holds a PhD in Computer Engineering from the University of Arizona in Tucson, Arizona. He was a Systems Engineer at Raytheon Missile Systems and then at Northrop Grumman Information Systems prior to joining Oak Ridge National Laboratory in 2005. He can be reached by email at nutarojj@ornl.gov.