On Constructing Optimistic Simulation Algorithms for the Discrete Event System Specification

James Nutaro
Oak Ridge National Laboratory

This paper describes a Time Warp simulation algorithm for discrete event models that are described in terms of the Discrete Event System Specification (DEVS). The paper shows how the total state transition and total output function of a DEVS atomic model can be transformed into an event processing procedure for a logical process. A specific Time Warp algorithm is constructed around this logical process, and it is shown that the algorithm correctly simulates a DEVS coupled model that consists entirely of interacting atomic models. The simulation algorithm is presented abstractly; it is intended to provide a basis for implementing efficient and scalable parallel algorithms that correctly simulate DEVS models.

Categories and Subject Descriptors: I.6.8 [**Simulation and Modeling**]: Type of Simulation—*Discrete event; Parallel*

General Terms: Algorithms, Design

Additional Key Words and Phrases: DEVS, parallel simulation, Time Warp, discrete-event simulation

1. INTRODUCTION

Time Warp algorithms for simulating DEVS models are relatively uncommon in the parallel discrete event simulation literature (see, e.g., [Zeigler et al. 1999; Glinsky and Wainer 2006; Liu 2007]), and simulations derived from existing software can fail to produce correct results. To reliably simulate a DEVS model, a Time Warp algorithm must address three issues. First, sequences of causally related events that occur in zero-time must be processed in the correct order [Nutaro 2003]. Second, causally independent inputs that arrive simultaneously at a model must be delivered to the model as a group, not individually [Chow and Zeigler 1994; Chow et al. 1994]. Third, the simulation algorithm must distinguish the three case of an internal event, external input, and simultaneous internal and external event [Chow and Zeigler 1994; Chow et al. 1994].

Author's address: James Nutaro, Oak Ridge National Laboratory, PO Box 2008, MS6085, Oak Ridge, TN 37831-6085. email:nutarojj@ornl.gov

The familiar PHOLD benchmark [Fujimoto 1990], suitably modified, illustrates these three issues. The modified PHOLD model comprises several servers that exchange tokens. Every server is contained in a logical process; tokens are processed in a first come, first served manner. Upon receiving a *single* token, the logical process proceeds in the usual way: the token is placed in the back of the queue and is processed if the server is idle or waits its turn if the server is busy. If the server simultaneously receives two or more tokens then all of the received tokens are discarded. If the server releases a token and receives a token at the same time then the server duplicates the released token.

The first rule, that single tokens be processed first come first served, presents no difficulty, but the last two rules are ambiguous. Consider a server whose service time for each token is 0 or 1 with equal probability. Suppose that the server begins at time zero without a token, receives two tokens at time 1, and receives a third token at time 1.5. For the sake of illustration, let's choose random numbers from the sequence 0, 0, 1, 0. The server can produce at least two different output sequences, corresponding to two different interpretations of the input sequence and model specification:

(1) a single token at time 1.5 or

(2) two tokens at time 1 and two tokens at time 2.

The problem is that the description of the model does not clearly state how to handle zero-time and simultaneous events. (Two papers, by Ronngren [Rönngren and Liljenstam 1999] and Wieland [Wieland 1997] respectively, summarize several solutions to the problem of zero-time and simultaneous events in parallel discrete event simulations.) This ambiguity can be removed in two steps: by formalizing the model in the terms of the Discrete Event System Specification and making explicit the super dense time base that is implicitly used by the DEVS abstract simulator [Nutaro and Sarjoughian 2004; Rönngren and Liljenstam 1999; Maler et al. 1992; Manna and Pneuli 1993]. These two steps impose a set of conditions on how the model is simulated; a logical process that satisfies these conditions can correctly simulate a DEVS atomic model.

This paper has three main parts. Section 2 introduces the essential elements of the Discrete Event System Specification and derives a logical process that can correctly simulate a DEVS atomic model. Section 3 uses this logical process to construct a specific Time Warp algorithm for simulating DEVS models that are closed and flat (i.e., that consist only of interacting atomic models). The remainder of this paper sketches a proof that the algorithm works correctly.

2. FROM ATOMIC MODELS TO LOGICAL PROCESSES

A DEVS atomic model is a structure

$$M =< X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$$

where

X is the set of inputs,
Y is the set of outputs,
S is the set of states,
$\delta_{ext} : Q \times X^b \rightarrow S$ is the external transition function,
where $Q = \{(s, \epsilon) \mid s \in S, 0 \leq \epsilon \leq ta(s)\}$ is the set of total states,
$\delta_{con} : S \times X^b \rightarrow S$ is the confluent transition function,
$\delta_{int} : S \rightarrow S$ is the internal transition function,
$\lambda : S \rightarrow Y$ is the output function, and
$ta : S \rightarrow \mathbb{R}_0^+$ is the time advance function.

The set $\mathbb{R}_0^+$ is the non-negative real numbers. A total state consists of a model state $s \in S$ and the time ϵ that has elapsed since the model entered that state. Inputs to the model are bags of events with elements from X; X^b denotes the set of these bags.

The parts of an atomic model are used to define the total state transition function and the total output function of a dynamic system (see, e.g., [Zeigler et al. 2000; Mesarovic and Takahara 1989]). This dynamic system has a super dense time base T. This time base is the set of non-negative real numbers crossed with the natural numbers; i.e.,

$$T = \mathbb{R}_0^+ \times \mathbb{N} \ .$$

T is ordered by

$$(t_1, c_1) < (t_2, c_2) \Leftrightarrow (t_1 < t_2 \vee (\ t_1 = t_2 \wedge c_1 < c_2))$$

and two elements are equivalent if $t_1 = t_2$ and $c_1 = c_2$, i.e.,

$$(t_1, c_1) = (t_2, c_2) \Leftrightarrow (t_1 = t_2 \wedge c_1 = c_2) \ .$$

Two elements of T are added with the rule

$$(t_1, c_1) \oplus (t_2, c_2) = \begin{cases} (t_1, c_1 + c_2) & \text{if } t_2 = 0 \\ (t_1 + t_2, 0) & \text{otherwise .} \end{cases}$$

Note that the $\oplus$ operator is not associative and it is not commutative. The function ϕ recovers the real part of an element (t, c); i.e.

$$\phi((t, c)) = t \ .$$

The purpose of the $\oplus$ operator is to distinguish simultaneous events from sequences of zero-time events. For example, if the last event at an atomic model happened at time (t_l, c_l) and its next event occurs immediately, then this event occurs at time $(t_l, c_l) \oplus (0, 1) = (t_l, c_l + 1)$. A subsequent zero-time event at the same process would occur at time $(t_l, c_l + 1) \oplus (0, 1) = (t_l, c_l + 2)$, but if the next event happens one second later then it occurs at time $(t_l, c_l) \oplus (1, 0) = (t_l + 1, 0)$. The second part of the time element orders sequences of instantaneous events.

The total state transition function Δ takes an initial total state $q_0 \in Q$ and an input trajectory $x[t_0, t_f)$ to a final state $q_f \in Q$. The input trajectory is a function

from the time base T to the set X^b (i.e., the set of bags with elements in X) and the non-event Φ; event trajectories, the only kind we are interested in, have a finite number of events in any interval $[t_0, t_f)$. The total output function Λ maps every total state to an output in Y. To summarize, the elements of an atomic model define a system whose parts are

$$
\begin{aligned}
&x[t_0, t_f) : \{t \in T | t_0 \leq t < t_f\} \to X^b \cup \{\Phi\}, \text{ an input trajectory,} \\
&\Delta : Q \times x[t_0, t_f) \to Q, \text{ the total state transition function, and} \\
&\Lambda : Q \to Y \text{ the total output function.}
\end{aligned}
$$

The value of the input trajectory $x[t_0, t_f)$ at any particular time $t \in [t_0, t_f)$ is written $x(t)$. Given an initial state $(s, \epsilon) \in Q$ and input trajectory $x[t_0, t_f)$, the state transition function recursively computes the state at time t_f (see [Zeigler et al. 2000; Nutaro and Sarjoughian 2004]) by

$$
\begin{aligned}
&\Delta[(s, \epsilon), x[t_0, t_f)] = \\
&1)\ \Delta[(s, ta(s)), x[t_0 \oplus (ta(s) - \epsilon, 0), t_f)] \\
&\text{if } \epsilon < ta(s) \wedge \forall t \in [t_0 \oplus (ta(s) - \epsilon, 0), t_f), x(t) = \Phi \\
&2)\ \Delta[(\delta_{int}(s), 0), x[t_0 \oplus (0, 1), t_f)] \\
&\text{if } \epsilon = ta(s) \wedge x(t_0) = \Phi \\
&3)\ \Delta[(\delta_{con}(s, x(t_0)), 0), x[t_0 \oplus (0, 1), t_f)] \\
&\text{if } \epsilon = ta(s) \wedge x(t_0) \neq \Phi \\
&4)\ \Delta[(\delta_{ext}(s, \phi(\tau) - \phi(t_0) + \epsilon, x(\tau)), 0), x[\tau \oplus (0, 1), t_f)] \\
&\text{where } \tau = \min\{t \in [t_0, t_f) | x(t) \neq \Phi\} \\
&\text{if } \tau \text{ exists and } \tau < t_0 \oplus (ta(s) - \epsilon, 0) \\
&5)\ (s, \phi(t_f) - \phi(t_0) + \epsilon) \\
&\text{if 1-4 do not apply.}
\end{aligned}
\tag{1}
$$

Note that case 5 applies if, at any step, x is not defined in the necessary interval. The total output function is

$$
\Lambda[(s, \epsilon)] = \begin{cases} \lambda(s) & \text{if } \epsilon = ta(s) \\ \Phi & \text{otherwise .} \end{cases} \tag{2}
$$

The total state transition and total output function collectively define four types of events (case 5 ends the simulation): output events, two types of self events, and external events. Case 1 is an output event; the simulation clock advances to the next self event time and, because $\epsilon = ta(s)$, this causes an output to occur. The time-stamp of the output event is equal to the time-stamp of the self event. Cases 2 and 3 are two instances of self events; the first instance is without a simultaneous input and the second is with a simultaneous input. Simultaneous input and self events have equivalent time-stamps. Case 4 is an external event; it processes the input events available at the event time.

Returning to the PHOLD example in section 1, the server is formally defined as an atomic model whose state is a pair $(q, \sigma) \in \mathbb{N} \times \mathbb{R}$; q is the number of tokens in the queue and σ is the time remaining to process the current token. The model's

and input and output set are $\{1\}$. The service time for each token is sampled from a discrete random variable r with two outcomes, 0 and 1, that are equally likely. The transition, output, and time advance functions are

$$\delta_{ext}((q,\sigma),e,x^b) = \begin{cases} (q+1,\sigma-e) & \text{if } q \neq 0 \text{ and } x^b \text{ has exactly one item} \\ (q,\sigma-e) & \text{if } q \neq 0 \text{ and } x^b \text{ has more than one item} \\ (1,r) & \text{if } q = 0 \text{ and } x^b \text{ has exactly one item} \\ (0,\infty) & \text{if } q = 0 \text{ and } x^b \text{ has more than one item} \end{cases}$$

$$\delta_{con}((q,\sigma),x^b) = \begin{cases} (q+1,r) & \text{if } x^b \text{ has exactly one item} \\ (q,r) & \text{if } x^b \text{ has more than one item} \end{cases}$$

$$\delta_{int}((q,\sigma)) = \begin{cases} (q-1,r) & \text{if } q > 1 \\ (0,\infty) & \text{otherwise} \end{cases}$$

$$\lambda((q,\sigma)) = 1$$

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

Suppose that the model's total initial state is $((0,\infty),0)$ and use the random number sequence $0,0,1,0$ when sampling r. To generate the two output sequences in section 1, we consider two input trajectories on the interval $[(0,0),(2.5,1))$:

$$x_1(t) = \begin{cases} \{1,1\} & \text{if } t = (1,0) \\ 1 & \text{if } t = (1.5,0) \\ \Phi & \text{otherwise} \end{cases}$$

and

$$x_2(t) = \begin{cases} 1 & \text{if } t = (1,0) \\ 1 & \text{if } t = (1,1) \\ 1 & \text{if } t = (1.5,0) \\ \Phi & \text{otherwise .} \end{cases}$$

The resulting state and output trajectories are computed with the total state transition and total output functions. Applying x_1 gives

$$\begin{aligned} &\Delta[((0,\infty),0),x_1[(0,0),(2.5,1))] \\ &= \Delta[((0,\infty),0),x_1[(1,1),(2.5,1))] \\ &= \Delta[((1,0),0),x_1[(1.5,1),(2.5,1))] \\ &= \Delta[((0,\infty),0),x_1[(1.5,2),(2.5,1))] \\ &= ((0,\infty),1) \end{aligned}$$

Table I. Simulations of the PHOLD Model using Input Trajectories x_1 and x_2

t	x_1	s	y	x_2	s	y
$(0,0)$	Φ	$(0,\infty)$	Φ	Φ	$(0,\infty)$	Φ
$(1,0)$	$\{1,1\}$	$(0,\infty)$	Φ	1	$(1,0)$	Φ
$(1,1)$				1	$(2,0)$	1
$(1,2)$				Φ	$(1,1)$	1
$(1.5,0)$	1	$(1,0)$	Φ	1	$(2,0.5)$	Φ
$(1.5,1)$	Φ	$(0,\infty)$	1			
$(2,0)$				Φ	$(1,0)$	1
$(2,1)$				Φ	$(0,\infty)$	1

and a single output event at time $(1.5, 1)$. Applying x_2 gives

$$\begin{aligned}
&\Delta[((0,\infty),0), x_1[(0,0),(2.5,1))] \\
&= \Delta[((1,0),0), x_1[(1,1),(2.5,1))] \\
&= \Delta[((2,0),0), x_1[(1,2),(2.5,1))] \\
&= \Delta[((1,1),0), x_1[(1,3),(2.5,1))] \\
&= \Delta[((2,0.5),0), x_1[(1.5,1),(2.5,1))] \\
&= \Delta[((2,0.5),0.5), x_1[(2,0),(2.5,1))] \\
&= \Delta[((1,0),0), x_1[(2,1),(2.5,1))] \\
&= \Delta[((0,\infty),0), x_1[(2,2),(2.5,1))] \\
&= ((0,\infty),0.5)
\end{aligned}$$

and output events at times $(1,1)$, $(1,2)$, $(2,0)$, and $(2,1)$.

The composition property of the state transition function (see, e.g., [Zeigler et al. 2000; Mesarovic and Takahara 1989]) lets us compute the same trajectories iteratively; the iterative algorithm is more familiar and easier to calculate with. The recursive calculations show the iterative steps when we begin with an elapsed time $\epsilon = 0$ and stop at the last event:

(1) Set the next event time t_N to the smaller of the next internal event time $t_{self} = t_0 \oplus (ta(s), 0)$ and the next input event time τ.
(2) If $t_{self} = t_N$ and $\tau < t_N$ then produce an output event with time-stamp t_{self} and compute the next state using δ_{int}.
(3) If $t_{self} = \tau = t_N$ then produce an output event with time-stamp t_{self} and compute the next state using δ_{con}.
(4) If $t_{self} < t_N$ and $\tau = t_N$ then compute the next state using δ_{ext}.
(5) Set the last event time t_0 to $t_N \oplus (0, 1)$.
(6) Repeat if there are more events.

The iterative algorithm gives the same state and output trajectories as the recursive function. Table I shows a simulation of the PHOLD model begun in the state $(0, \infty)$ and fed the input trajectories x_1 and x_2.

The role of the super dense time base is readily apparent: simultaneous events have equal time-stamps and sequences of zero-time events are ordered by the clock's second field. These two properties are sufficient to ensure that the local causality constraint, which is defined in terms of time-stamp ordering (see, e.g., [Fujimoto

2000]), preserves input-output causality, which is defined in terms of the total state transition function and total output function (see, e.g., [Mesarovic and Takahara 1989]). This is essential for using well-known parallel discrete event simulation algorithms to correctly simulate DEVS models [Nutaro and Sarjoughian 2004; Nutaro 2003].

The iterative algorithm defines the event handling procedure for a logical process. The logical process uses the super dense simulation clock to time-stamp self events and output events. The order of the time-stamps unambiguously reflects the model's input-output causality. Input and self events with equal time-stamps require that 1) the input and self events be formed into a single super-event for processing by the confluent or external transition function and 2) processed self and input events with the same time-stamp be rolled back and then incorporated into the new super-event. This does not require that an output event with the same time-stamp be rolled back! Input and output events with identical time-stamps are causally independent. (This is true because every DEVS model defined on a super dense time base is strongly causal [Mesarovic and Takahara 1989] and, equivalently, has a look-ahead of $(0, 1)$.)

The algorithm presented in section 3 uses a computational representation of an atomic model [Nutaro and Sarjoughian 2004]. This representation preserves the input-output behavior of the atomic model and presents a logical process like view by treating input, output, and self events uniformly. The computational representation consists of a set of input values, a set of output values, and a set of time-stamps for those values. An event is a (time,value) pair. The event e and pair (t, v) are used interchangeably. Events are ordered by their time-stamps.

Super-events are produced with event concatenation. If e_1 and e_2 are two events with equal time-stamps then they can be concatenated to form a new event e_3 whose value is the bag union of the values of e_1 and e_2. Event concatenation, denoted by $\cdot$, is defined by

$$(t, \{a_1, a_2, ..., a_n\}) \cdot (t, \{b_1, b_2, ..., b_m\}) = (t, \{a_1, ..., a_n, b_1, ..., b_m\}) \ .$$

Concatenation is associative and commutative.

The current state of an atomic model is determined by its input history; the computational representation deals only with this input history. The next self event and output event are determined from the model's input history. Formally, the input history is a stack (sometimes referred to as a sequence) of events that is operated on by the structure

$$STC =< E, G, push, eN, out >$$

where

E is the event set,
G is the set of event stacks $(e_0 e_1 ...)$ where $e_i \in E$
$push : G \times E \to G$ is the transition function,
$eN : G \to E$ is the next event function,
$out : G \to E$ is the next output function,

Table II. Logical Process Calculation of the PHOLD Model using Input Trajectory x_1

t	x_1	s	$eN(s)$	$out(s)$
$(0,0)$		$\emptyset$	$((\infty,0),\zeta)$	$((\infty,0),\Phi)$
$(1,0)$	$e_1=((1,0),\{1,1\})$	$\emptyset e_1$	$((\infty,0),\zeta)$	$((\infty,0),\Phi)$
$(1.5,0)$	$e_2=((1.5,0),1)$	$\emptyset e_1 e_2$	$\zeta_1=((1.5,1),\zeta)$	$((1.5,1),1)$
$(1.5,1)$		$\emptyset e_1 e_2 \zeta_1)$	$((\infty,0),\zeta)$	$((\infty,0),\Phi)$

and the push function is defined by

$$push((e_0 e_1 ... e_n), e) = e_0 e_1 ... e_n e \ .$$

The *push* function causes the logical process to execute an event and, equivalently, the atomic model to compute one of its transition functions. After pushing an event, the logical process executes line 5 of the iterative algorithm, thereby advancing the simulation clock and causing the next self and output events to have strictly larger time-stamps than the last event. The eN function gives the next self event that the logical process will execute and, equivalently, the atomic model's next internal event. The *out* function gives the next event that the logical process will send to other logical processes and, equivalently, the next output event from the atomic model.

The initial state of the atomic model is represented by a stack, called the empty stack, which contains a single initialization event whose time-stamp is $(0,0)$. The logical process begins its execution at step 5 of the iterative algorithm, effectively starting the simulation at time $(0,1)$. This presents no practical difficulties if the logical processes only receive events from each other (i.e., there are no external agencies that can inject events into a running simulation), and it has two useful consequences:

$$e_n < eN(e_0 e_1 ... e_n) \text{ , and}$$
$$e_n < out(e_0 e_1 ... e_n) \ .$$

These two properties are due to the fifth step in the iterative algorithm which always advances the simulation clock by $(0,1)$. Notice, however, that $eN(e_0 e_1 ... e_n)$ and $out(e_0 e_1 ... e_n)$ have equal time-stamps because output and internal events always coincide. Input sequences that do not result in an output event (i.e. $\lambda(s) = \Phi$) have $out(e_0 e_1 ... e_n) = (t, \Phi)$.

The PHOLD example will, once again, illustrate the use of this structure. Input values are taken from the set X^b, output values from the set Y, the value ζ indicates a self event, and $\emptyset$ denotes the initial event. Table II and Table III show how a logical process executes the input trajectories x_1 and x_2.

3. A TIME WARP ALGORITHM FOR DEVS MODELS

This algorithm is based on similar DEVS simulation algorithms presented in [Christensen 1990] and [Zeigler et al. 2000]. It differs from these algorithms chiefly in its use of the super dense time base and, consequently, its guarantee to correctly simulate every model defined in terms of the Discrete Event System Specification.

Consider a set of processes labeled $1, 2, 3, ..., N$. Processes communicate by exchanging messages $(proc, e)$, where e is an event and *proc* is the label of the origi-

Table III. Logical Process Calculation of the PHOLD Model using Input Trajectory x_2

t	x_1	s	$eN(s)$	$out(s)$
$(0,0)$		$\emptyset$	$((\infty,0),\zeta)$	$((\infty,0),\Phi)$
$(1,0)$	$e_1=((1,0),1)$	$\emptyset e_1$	$\zeta_1=((1,1),\zeta)$	$((1,1),1)$
$(1,1)$	$e_2=((1,1),1)$	$\emptyset(e_2\cdot\zeta_1)$	$\zeta_2=((1,2),\zeta)$	$((1,2),1)$
$(1,2)$		$\emptyset e_1(e_2\cdot\zeta_1)\zeta_2$	$\zeta_3=((2,0),\zeta)$	$((2,0),1)$
$(1.5,0)$	$e_3=((1.5,0),1)$	$\emptyset e_1(e_2\cdot\zeta_2)e_3$	$\zeta_3=((2,0),\zeta)$	$((2,0),1)$
$(2,0)$		$\emptyset e_1(e_2\cdot\zeta_2)e_3\zeta_3$	$\zeta_4=((2,1),\zeta)$	$((2,1),1)$
$(2,1)$		$\emptyset e_1(e_2\cdot\zeta_2)e_3\zeta_3\zeta_4$	$((\infty,0),\zeta)$	$((\infty,0),\Phi)$

nating process. All events are time-stamped as discussed in section 2. Each process has

(1) a bag A that contains messages the process has received but not processed,
(2) a bag U that contains messages the process has received and executed their events,
(3) an event sequence s that is the current input history of the model being simulated, and
(4) a set S of event sequences that are the process check-points.

The time-stamp of the event e_n in the event sequence $s=e_1e_2...e_n$ is denoted by $s.t$. The smallest time-stamp in the bag A is denoted by $min\ A$ and the largest time-stamp by $max\ A$; $min\ U$ and $max\ U$ are similarly defined. The event sequence $s\in S$ with the smallest $s.t$ is denoted by $min\ S$.

The symbol $\emptyset$ denotes an empty bag, set, or sequence. The meaning should be clear from the context in which the symbol is used. The event r is used to denote a time-stamped event with a special value that means 'rollback to time t', where t is the event's time-stamp.

First in-first out (FIFO) channels are used for inter-process communication. The function *receive* returns the next message that is available to the process, or the special symbol *blank* if no message is available. The function $send(m)$ sends message m to the appropriate set of processes. This set is defined by the system coupling, which is not represented explicitly. Rather, it is assumed that some fixed system coupling is known prior to beginning a simulation (i.e., $send(m)$ always sends the message m to the same set of processes. (The *send* function acts like the coupling functions of a DEVS coupled model. It can duplicate events, eliminate events, and change the value of an event as required by the model's coupling specification, but the *send* function can not alter the time-stamp of any event.) Output events with the value Φ are not transmitted; the *send* function discards them. The *send* and *receive* functions are a typical model of a FIFO communication channel.

The DEVS Time Warp algorithm is listed as Algorithm 1. The algorithm is run by each process involved in the computation. The statement $m.event$, where m is a message, refers to the event associated with message m. The statement $m.t$ refers to the time-stamp assigned to $m.event$. Similarly, for an event e, $e.t$ refers to the time-stamp assigned to the event. The statement $m.proc$ refers to the process that sent the message m.

The algorithm works as follows. At the beginning of each iteration, check for any available messages. If a message is available, check to see if it is a rollback indicator.

Algorithm 1 Time Warp algorithm for DEVS models.

1: $s \leftarrow \emptyset$, $S \leftarrow \{s\}$ {The initial event history is empty and it is saved}
2: $A \leftarrow \emptyset$, $U \leftarrow \emptyset$ {The available and used input bags are initially empty}
3: $lr \leftarrow (0, 0)$ {The last local rollback time}
4: **while** terminating condition is not met **do**
5: $msg \leftarrow receive$ {Get the next message}
6: **if** $msg \neq blank$ **then** {Was a message was found?}
7: **if** $msg.event = r$ **then** {If rollback, discard msgs from the sender}
8: $A \leftarrow A - \{(proc, x) | proc = msg.proc \wedge msg.t \leq x.t \wedge (proc, x) \in A\}$
9: $U \leftarrow U - \{(proc, x) | proc = msg.proc \wedge msg.t \leq x.t \wedge (proc, x) \in U\}$
10: **end if**
11: **if** $msg.t \leq s.t$ **then** {Does the message require a local rollback?}
12: $lr \leftarrow msg.t$
13: **if** there exists a check-point $z \in S$ such that $z.t > msg.t$ **then**
14: $send(r)$ with $r.t$ equal to the smallest such $z.t$
15: **end if**
16: $S \leftarrow S - \{z | z \in S \wedge z.t \geq msg.t\}$ {Discard useless checkpoints}
17: $T \leftarrow \{(proc, x) | x.t > (max\ S).t \wedge (proc, x) \in U\}$
18: $U \leftarrow U - T$ {Remove newly available messages from the used bag}
19: $s \leftarrow max\ S$ {Rollback the state}
20: $A \leftarrow A \cup T$ {Add newly available messages to the available bag}
21: **end if**
22: **if** $msg.event \neq r$ **then** {Add the received event to the available bag}
23: $A \leftarrow A \cup \{msg\}$
24: **end if**
25: **end if**
26: $\Gamma \leftarrow \{a_1, a_2, ..., a_n | a_i \in A \wedge a.t = min\ A\}$ {Get input events with the smallest time-stamps}
27: $x \leftarrow a_1 \cdot a_2 \cdot ... \cdot a_n$ {Create a single event from these input events}
28: **if** $(\Gamma = \emptyset \vee x.t \geq out(s).t) \wedge out(s).t > lr$ **then** {Send an output?}
29: $send(out(s))$
30: **end if**
31: **if** $x.t = eN(s).t$ **then** {Confluent event?}
32: $A \leftarrow A - \Gamma$
33: $s \leftarrow push(s, x \cdot eN(s))$
34: $U \leftarrow U \cup \Gamma$
35: **else if** $x.t < eN(s).t$ **then** {External event?}
36: $A \leftarrow A - \Gamma$
37: $s \leftarrow push(s, x)$
38: $U \leftarrow U \cup \Gamma$
39: **else** {Otherwise, it is an internal event}
40: $s \leftarrow push(s, eN(s))$
41: **end if**
42: $S \leftarrow S \cup \{s\}$ {Save the model state}
43: **end while**

Table IV. Table of Commonly Used Symbols

Symbol	Interpretation
U	Bag of messages that have been processed
A	Bag of messages that have not been processed
S	Set of event sequences
$e.t$	Time stamp of an event e
$msg.t$	Time stamp of the event carried by message msg
$s.t$	Time stamp of the last event in the sequence s
r	A roll back message
$blank$	An empty message, absence of a message
$\emptyset$	An empty bag, set, or sequence (depending on the context)
GVT	Global virtual time
$min\ A$, $min\ U$	The smallest message time-stamp in A, U
$min\ S$	The event sequence in S with the smallest $s.t$
$(min\ S).t$	The time-stamp of the last event in $min\ S$
lr	The last local rollback time

If so, remove from the bag of available and used messages all messages that have a time-stamp less than or equal to the rollback time-stamp and that were sent by the process that created the rollback message. This has the effect of canceling incorrect messages that were previously received from the creator of the rollback message.

Next, check to see if the received message (rollback or otherwise) is in the simulated future. If it is, then add the message to the bag of messages that are available to be processed. Otherwise, restore the model to a checkpoint state just prior to the message time-stamp. This includes restoration of the model state, the available message bag, and the used message bag.

A rollback notification is sent if there is a checkpoint with a time-stamp larger than the received message (see section 2). If this is the case, then the process undergoes a regular rollback; it will need to recall, recompute, and resend output events that it transmitted after the rollback time. Otherwise only a local rollback is needed. Any output message that was sent at the rollback time is correct, but the state of the process at the rollback time must be recomputed. If a rollback message is sent, it informs the process's neighbors that messages from it with a time-stamp greater than or equal to the rollback time should be discarded. Finally, the received message is added to the bag of available messages.

Lines 26 through 41 compute the next state of the DEVS model. To do this, first check the collection of available inputs. If input is available, then construct a single external event by concatenating the available events with time-stamps that are equal to $min\ A$. If this time-stamp is greater than or equal to the model's time of next event and the process is not undergoing a local rollback at the same simulation time, then compute and send the model's output. Next, the model state is updated as appropriate (i.e., using a confluent, external, or internal transition). If a confluent or external transition is executed then the processed inputs are moved to the used event bag. At line 42, the model state is saved and we repeat.

4. CANCELING EVENT SEGMENTS

The algorithm uses implicit anti-messages to cancel sequences of output events. The batch cancellation used here differs from aggressive and lazy cancellation schemes,

in which explicit anti-messages are used to undo individual events (see, e.g., [Rajan and Wilsey 1995; Fujimoto 2000; Ferscha 1995]). It is more closely related to batch cancellation schemes described in [Xu and Tropper 2005; Zeng et al. 2004; Martin et al. 1999]. The cancellation scheme works on the principle that the output trajectory of a DEVS model is causally dependent on its past input trajectory. Causal dependence is defined in terms of the input-output function that is realized by the DEVS model, with the simulation time-stamps being suitably augmented for detecting this type of dependency (see section 2).

Suppose that a model has speculatively processed a sequence of events with time-stamps up to time t. If an event arrives with a time-stamp earlier than t, say at time $t' < t$, then the outputs produced by the model with time-stamps greater than t' are possibly incorrect and must be canceled. To cancel them, it is sufficient to send a single message which indicates that all output events produced by that model after time t' are possibly incorrect and should be discarded. If FIFO channels are used, then cancellation messages cannot overtake subsequent output events. Consequently, the scheme preserves causality.

To illustrate the scheme, consider the event sequence shown in Fig. 1. Suppose the process that produced these output events receives an anti-message with time-stamp $t_{rollback}$. An aggressive cancellation scheme will send three anti-messages, one for each output event occurring after $t_{rollback}$. A lazy cancellation scheme will send between 0 and 3 anti-messages, depending on how the computation proceeds following the rollback. The implicit anti-message scheme will send one anti-message that cancels all events in the interval $[t_{rollback}, (\infty, \infty))$.

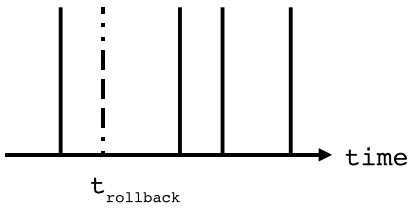

Fig. 1. Canceling an event segment.

5. FOSSIL COLLECTION

To support rollbacks, the algorithm constructs a series of checkpoints that record the process's state and input history. Inevitably, the algorithm will exhaust the available memory unless a scheme is in place for finding and deleting checkpoints that are no longer useful. This scheme is called fossil collection. Fossil collection for time warp systems has been an active area of research (see, e.g., [Soliman 1999; Young et al. 1998; Bruce 1995]). Algorithm 1 can be used in conjunction with any convenient fossil collection scheme.

A simple fossil collection scheme can illustrate the basic concepts ([Mattern 1993] provides an overview of the essential algorithms). This method first computes an approximation of the global virtual time, and then uses this approximation to find and delete unneeded checkpoints.

The global virtual time approximation starts by finding a consistent cut. This cut divides the global event set into two partitions; a past partition that contains only events which were processed prior to the cut, and a future partition of events that have been (or will be) processed after the cut. Once these partitions have been found, the global virtual time is approximated by the minimum event time-stamp in the future partition.

Each process retains one checkpoint whose time-stamp is less than the global virtual time approximation. All other checkpoints with time-stamps that are less than that of the retained checkpoint are discarded. The retained checkpoint makes it possible to roll the process back to a consistent state (i.e., a state from which no further rollbacks will be required).

6. PROOF OF CORRECTNESS

The correctness proof has two parts. The first part is a safety proof which shows that the algorithm is correct up to global virtual time. That is, if global virtual time has advanced to (t, c) then each process has constructed the correct input and output sequences up to time (t, c). The second half is a liveness proof which shows that global virtual time increases. The logical dependencies between the theorems that precede the final correctness demonstration are shown in Fig. 2.

The safety proof is constructed as follows. First, it is shown that there always exists a checkpoint sufficiently far in the past that rollbacks can be executed. A simple proof is presented first. A more complex proof, which allows for fossil collection, is also constructed.

Having proved that rollbacks can always be processed, a pair of causality theorems are presented. Taken together, these state that events are processed in time-stamp order. This property, in conjunction with the check pointing theorem, is needed to prove a pair of retention theorems. The retention theorems state that all correct input events, and no incorrect input events, are retained up to global virtual time. The sequencing theorem is proved next, and it states that the algorithm produces correct output events given correct input events. The conclusion that the algorithm is correct up to global virtual time follows immediately from the preceding theorems.

The liveness proof is taken, with minor adaptations, from [Leivent and Watro 1993], where it was shown that if certain progress axioms hold, then global virtual time always eventually increases.

7. CHECK-POINTING THEOREM

The check-pointing theorem states that it is always possible to perform the rollback step at line 19. This is done by showing that S always contains a sequence z such that $z.t$ is less than the smallest time-stamp that can be received from another process. By definition, the time-stamp of any event received by a process is greater than or equal to global virtual time (denoted GVT). So it is sufficient that the algorithm keep at least one checkpoint whose time-stamp is less than GVT.

THEOREM 7.1 CHECKPOINTING THEOREM. *If GVT is non-decreasing, then S is not empty and $(min\ S).t < GVT$.*

A simple proof can be constructed.

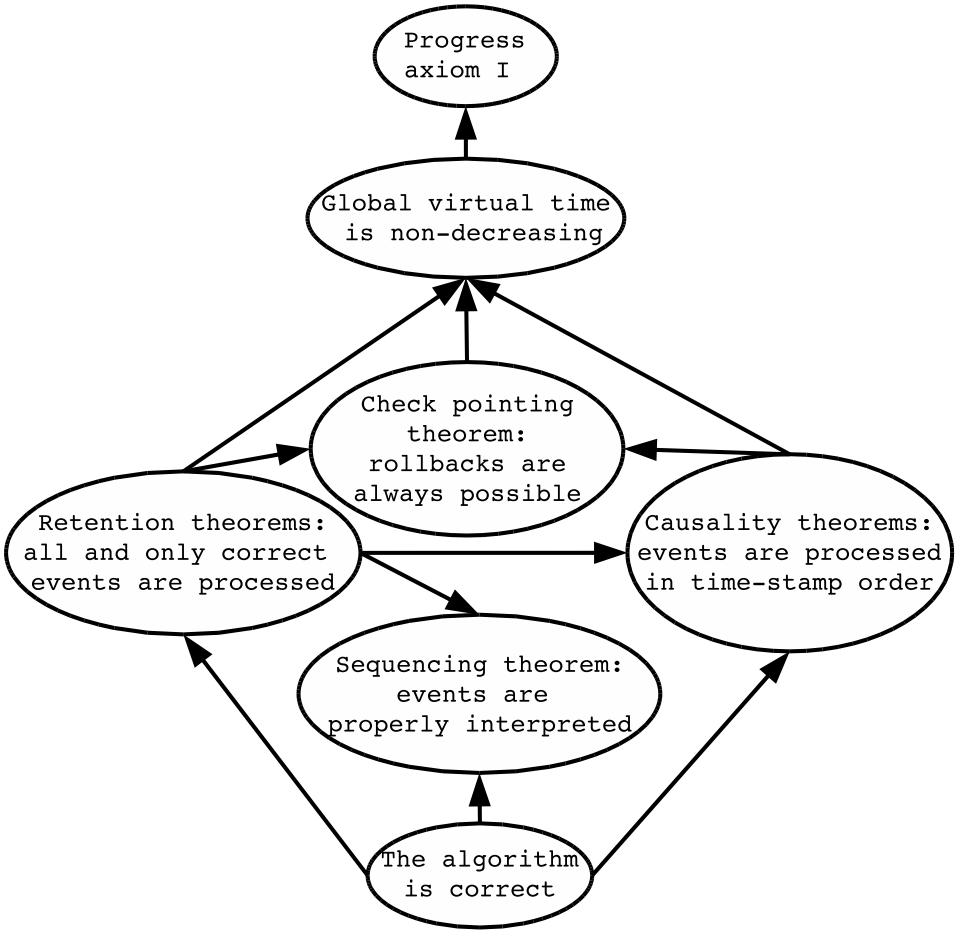

Fig. 2. Logical dependency of the intermediate theorems in the safety proof.

PROOF. Initially, every process's saved state set S contains the empty sequence, and so $(min\ S).t = (0,0)$. GVT is, consequently, greater than $(0,0)$. At every process, the current sequence s must be such that $(0,0) \leq s.t$. Since $out(s).t > s.t$, it must be true that $out(s).t > (0,0)$. A process can send either $out(s)$ or a rollback event r, both of which must have time-stamps greater than $(0,0)$. Consequently, any message received by any process will contain a time-stamp that is greater than $(0,0)$. It follows that the initial sequence can not be removed from S at line 16. □

If a fossil collection scheme is introduced, then this proof is no longer appropriate because it relies on keeping the initial checkpoint forever. A somewhat more complicated proof allows for correctness so long as the fossil collection scheme is similar to the one described in section 5.

PROOF. The theorem holds prior to the first iteration of the algorithm, as was shown in the first proof. Now, suppose the theorem holds at line 4 after an arbitrary number of iterations. Consider the next iteration of the algorithm. Since GVT is

assumed to be non-decreasing, if S is unchanged then the theorem continues to hold. It is sufficient to consider only those lines where S is altered. These are lines 16 and 42.

Prior to executing line 16, the induction hypothesis requires at least one sequence $z \in S$ such that $z.t < GVT$. This element is not removed when line 16 is executed, so S contains at least one element after executing line 16.

At line 42, a sequence is added to S that is the result of appending to s a previously unprocessed event. By definition, this event had a time-stamp greater than or equal to GVT. The induction hypothesis requires that $(min\ S).t < GVT$ prior to executing line 42. Clearly, $(min\ S).t$ can not increase as a result of executing line 42. So $(min\ S).t < GVT$ must hold after executing line 42. Therefore, S contains at least one element with time-stamp less than GVT after executing line 42. At no other point is S altered. □

8. CAUSALITY THEOREMS

There are two causality theorems. The first states that, if any events are marked as processed (i.e., are in the used event bag U), then these events have time-stamps in the past. The second causality theorem states that all unprocessed events (i.e., are in the available event bag A) are in the future. Taken together, the causality theorems state that events are processed in time-stamp order. The retention and sequencing theorems will complete the safety proof by showing that the correct events are processed in time-stamp order.

THEOREM 8.1 FIRST CAUSALITY THEOREM. *Suppose GVT is non-decreasing. If $U \neq \emptyset$, then $max\ U \leq s.t$.*

PROOF. Let e be the largest event (i.e., with the largest time-stamp) in U. This event must have been added to U at lines 34 or 38. In both cases, e was concatenated with zero or more events that have identical time-stamps before being pushed onto s. So it must be that $e.t = s.t$ after executing line 34 or line 38. Since e is the largest element in U, $max\ U \leq s.t$.

At lines 9 and 18, elements are removed from U. So either $max\ U$ stays the same, is reduced, or U becomes empty. If this is the first removal of elements from U, then the theorem must have held before executing line 9 or 18. Since $s.t$ remains the same and $max\ U$ does not grow or U becomes empty, the theorem must remain true after executing lines 9 or 18. It follows that if U is only altered at lines 34, 38, 9, and 18, then the theorem will always holds so long as $s.t$ is not altered.

Increasing $s.t$ without altering U, as is done at line 40, will not effect the truth of the theorem. That $s.t$ increases is a consequence of the second causality theorem (which is yet to be proved) and the fact that $eN(s).t > s.t$. This leaves line 19 where $s.t$ is, possibly, reduced. At line 18, all messages with time-stamps greater than $(max\ S).t$ are removed from U. At line 19, s is set to $max\ S$. So it must be that $max\ U \leq (max\ S).t = s.t$, and so $U \leq s.t$. □

THEOREM 8.2 SECOND CAUSALITY THEOREM. *Suppose GVT is non-decreasing. If $A \neq \emptyset$, then $min\ A > s.t$.*

PROOF. Consider the first iteration of the algorithm. Initially, A is empty. The first message added to A must occur at lines 20 or 23. If $msg.t > s.t$, then line

20 is not executed and at line 23 msg is added to A. This is the only message in A, and so $min\ A > s.t$. If $msg.t \leq s.t$, then line 20 is executed. At line 20, the elements of T, all of which are larger than $s.t$, are added to A. So A is empty or $min\ A > s.t$ after executing line 20.

The last possibility is that msg is $blank$, in which case no message is added to A. In any event, the theorem holds after reaching line 26. To show that the theorem holds for the remainder of the iteration, it is sufficient to show that the time-stamp of the message in A is greater than $s.t$. Suppose it is not. Then a rollback must have occurred at line 11. After the rollback, $s.t < msg.t$. If line 22 evaluates to true, then $s.t < msg.t = min\ A$. This contradicts the assumption that $min\ A \leq s.t$. Otherwise, A is empty and the theorem is vacuously true.

Having established the truth of the theorem on the first iteration, suppose it holds at line 4 after an arbitrary number of iterations. Consider the next iteration. At line 5, the process receives a message msg or $blank$. If $blank$, the theorem must still hold upon reaching line 26 since neither A nor s are altered.

Now, suppose $msg.event = r$. Executing line 8 either makes A an empty bag or $min\ A$ is not decreased. This follows from the fact that all removed messages have time-stamps greater than or equal to the time-stamp of the rollback message. So at line 11, the theorem still holds.

Next, suppose that line 11 evaluates to true. At line 19, s is altered, causing $s.t < msg.t$ to hold. At line 20, messages with time-stamps larger than $s.t$ are added to A. So the theorem still holds at line 22.

Finally, at line 23 it is impossible to add a message with time-stamp less or equal to $s.t$ to A. If line 11 evaluated to false, then clearly $msg.t > s.t$. Otherwise, the process is rolled back to at state with $s.t < msg.t$. Consequently, at line 23, the message added to A has a time-stamp greater than $s.t$.

Having safely reached line 26, it is sufficient to show that $x.t$ is greater than $s.t$ at line 27. It has been established that $min\ A > s.t$. Note that x is the result of concatenating the events in A with time-stamps equal to $min\ A$. It follows that $x.t = min\ A$. So $s.t < x.t$ as desired. □

9. RETENTION THEOREMS

The retention theorems state that all correct events, and no incorrect events, with time-stamps less than GVT have been processed. In conjunction with the causality theorems, this means that the correct events have been processed and they have been processed in the correct order. It will only remain to show that the events have been properly interpreted with respect to the DEVS formalism. This is the subject of the sequencing theorem, which will be considered last.

THEOREM 9.1 FIRST RETENTION THEOREM. *Select an arbitrary time $T < GVT$ and assume that GVT is non-decreasing. Then the bag A contains no events with time-stamps less than or equal to T.*

PROOF. If A is empty, the theorem is obviously true. Otherwise, it is known from theorem 8.2 that $min\ A > s.t$. If $s.t > GVT$ then the theorem is true because $min\ A > s.t > GVT$. If $s.t \leq GVT$, then either A contains the smallest unprocessed event and so $min\ A = GVT$ or $min\ A > GVT$. In either case the theorem holds. □

THEOREM 9.2 SECOND RETENTION THEOREM. *Select an arbitrary time $T < GVT$ and assume that GVT is non-decreasing. Further assume that if only correct input events are consumed by the model (i.e., placed into the bag U) then only correct output events are produced by the model. Lastly, assume that the messages received by a process contain all of the correct inputs for that process up to GVT. Then the bag U contains all correct input events with time-stamp less than or equal to T and no such incorrect events.*

PROOF. The proof proceeds by induction on the total (i.e., for all processes) number of events processed and messages received. The theorem is clearly true when no events have been processed and no messages have been received. Suppose that the theorem holds when n events have been processed and messages received.

If the next activity to occur is the receipt of a message, then the time-stamp associated with that message is, by definition, greater than or equal to GVT. If that message causes a rollback, then only events with time-stamps greater than or equal to GVT will be removed from U at lines 9 and 18. Consequently, the theorem continues to hold at line 25.

Suppose instead that the $(n+1)^{\text{st}}$ activity is the processing of an event. The event must have a time-stamp greater than or equal to GVT (due to theorem 8.2). If the time-stamp of the event is greater than GVT or if the event is a self event, then the theorem continues to hold by virtue of the induction hypothesis. If the time-stamp of the event is equal to GVT and it is an external event, then the correctness of the event follows from the induction hypothesis (i.e., the event is the output of a process that has consumed only correct events and has received all of the correct inputs that causally preceded that output). So the theorem holds in this case as well. □

10. SEQUENCING THEOREM

The sequencing theorem has two purposes. First, it validates the correct output assumption stated in the second retention theorem. Second, in combination with the second retention theorem, it shows that the correct input and output sequences are generated by the algorithm.

For the moment, assume the available message bag A contains messages that describe a correct event trajectory. To apply this event trajectory to the model, the algorithm is run by itself (i.e., only a single process is used), thereby assuring that no input messages can be received (or, equivalently, remove lines 5 to 25). Then the sequencing theorem is as follows.

THEOREM 10.1 SEQUENCING THEOREM. *Consider the algorithm with lines 5 to 25 removed, and assume the bag A contains exactly the correct input event sequence up to time T. Then the algorithm produces only correct outputs up to time T.*

PROOF. The removal of lines 5 to 25 yields an algorithm for applying the input trajectory contained in A to the model. This algorithm is identical to the algorithm developed in [Nutaro and Sarjoughian 2004]. In the same paper, this algorithm is shown to produce the correct state and output trajectories for DEVS atomic models. □

The sequencing theorem justifies the assumption made in the second retention

theorem concerning correct outputs. This leads to the correct event property, which was the original goal.

THEOREM 10.2 SAFETY THEOREM. *The algorithm is correct up to time* $T < GVT$.

PROOF. This theorem is an immediate consequence of the sequencing theorem and second retention theorem. □

The safety theorem states that the algorithm will generate the input and output trajectories that are correct for a model that is described in the terms of the DEVS formalism. The safety theorem was built in three parts. First, events are processed in time-stamp order. Second, only and all correct event are processed. Third, those events are interpreted correctly with respect to the DEVS formalism (i.e., as internal, external, and confluent events).

11. LIVENESS THEOREMS

The hypothesis of the safety theorem includes the assumption that global virtual time is non-decreasing. To prove this assumption, it is sufficient show that the progress axioms stated by Leivent and Watro [Leivent and Watro 1993] hold. The progress axioms are

(1) For all processes, a rollback step occurs if, and only if, the process has received an input message that is an anti-message or has a time-stamp that is in the past (such a message is called a *thorn*).
(2) A process incorporates (i.e., processes) a message if, and only if, it has no thorns and has a message to process.
(3) If a process incorporates a message, then the message time-stamp is the least time-stamp of the messages available for the process to incorporate.
(4) The output function requirement holds.

The output function requirement is as follows [Leivent and Watro 1993]: A process is modeled as a function from an initial state and sequence of inputs messages to a sequence of output messages such that the time stamp assigned to an output must be strictly larger than the time stamps of any preceding inputs.

The output function requirement was demonstrated in section 2. That progress axiom 1 holds is clear from the algorithm text, and lines 13 and 28 in particular. Progress axiom 2 is apparent at lines 18 and 20, where messages with time-stamps in the future are moved from the used message bag U to the available message bag A. Progress axiom 3 is apparent from lines 26 and 27, where the message that will be incorporated is constructed.

Lemma 11.1 states that GVT is non-decreasing. In [Leivent and Watro 1993], the lemma is stated in terms of a computation performed by a computational system. The proof of the lemma relies only on progress axiom 1. Only the properties of the computational system itself are considered here (the algorithm is a computational system; see [Leivent and Watro 1993] for the details). Lemma 11.1, appropriately paraphrased, is as follows:

LEMMA 11.1. *If a computational system satisfies progress axiom 1, then GVT in a computation performed by the computational system is non-decreasing.*

Lemma 11.1 satisfies the hypothesis of the correct event theorem (i.e., that GVT is non-decreasing). Theorem 11.2 [Leivent and Watro 1993] shows that actual forward progress is inevitable.

An *error free* run is a computation performed by a computational system that does not enter an error state during a rollback free computation. The safety theorem, in conjunction with progress axiom 1, ensures that the algorithm generates error free runs (i.e., it is an error-free computational system). A *terminal* state is one in which no processes have messages to incorporate. When simulating DEVS models, this occurs when all of the models are passive. A *fair run* is a computation in which all messages sent are eventually received, and all processes have the opportunity to process the messages they receive.

THEOREM 11.2 LIVENESS THEOREM. *Assume an error-free computational system with non-decreasing input histories. If a fair run satisfies the progress axioms, then either it reaches a terminal state or GVT is always eventually increasing.*

Theorem 11.2 shows that the algorithm will reach a state in which all of the models are passive or, given any desired simulation end time, the algorithm will reach it.

12. CORRECT SIMULATION OF DEVS MODELS

The DEVS Time Warp algorithm correctly simulates a DEVS model if the algorithm exhibits the correct event property (see [Nutaro and Sarjoughian 2004]) and the simulation time always eventually increases. Both of these properties have been shown to hold for algorithm 1 and so

THEOREM 12.1. *Algorithm 1 correctly simulates DEVS models.*

PROOF. This follows immediately from the safety and liveness theorems. □

13. CONCLUSIONS

The correctness proof relies on the separation of the model and simulator that are intrinsic to DEVS (and mathematical systems formalisms in general). Theorem 12.1 only states that, given a model expressed as a DEVS, the algorithm will produce the same input-output behavior as the DEVS abstract simulator, up to the global virtual time. The argument relies in no way on information about the model to be simulated (except, of course, that it be a DEVS). The algorithm will generate correct behavior for any DEVS model, but in no way guarantees or requires that the model itself be valid or properly implemented.

Algorithm 1 is a skeleton on which high performance DEVS simulation environments can be built. In an implementation, the choice of state saving and restoration techniques, global virtual time calculation algorithm, targeted machine architecture, and targeted model types will all have a significant effect on performance. The correctness argument is simplified by the batch based cancellation scheme, but it could be replaced with other schemes. Future research in parallel algorithms for DEVS models must address these performance issues within the strictures of correctly simulating DEVS models.

Conversely, section 2 describes the basic steps for converting a logical process simulator into a simulator that correctly executes DEVS models. The conversion

process has three basic steps: changing the simulation clock from the real numbers to the super dense time base T; modifying the logical process's event handler to recognize confluent, internal, and external events; and changing the rollback rule to correctly construct a single super-event from simultaneous input and self events. This is a promising approach that builds on the decades of research on high performance simulation software.

REFERENCES

Bruce, D. 1995. The treatment of state in optimistic systems. In *PADS '95: Proceedings of the ninth workshop on Parallel and distributed simulation*. IEEE Computer Society, Washington, DC, USA, 40–49.

Chow, A., Zeigler, B., and Kim, D. H. 1994. Abstract simulator for the parallel DEVS formalism. In *Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*. IEEE Press, Piscataway, NJ, USA, 157–163.

Chow, A. C. H. and Zeigler, B. P. 1994. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference*. Society for Computer Simulation International, San Diego, CA, USA, 716–722.

Christensen, E. 1990. Hierarchical Optimistic Distributed Simulation: Combining DEVS and Time Warp. Ph.D. thesis, University of Arizona, Department of Electrical and Computer Engineering.

Ferscha, A. 1995. Parallel and Distributed Simulation of Discrete Event Systems. In *Handbook of Parallel and Distributed Computing*. McGraw-Hill, Inc., New York, NY.

Fujimoto, R. 1990. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*. Vol. 22. Society for Computer Simulation, San Diego, California, USA, 23–28.

Fujimoto, R. 2000. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, Inc., New York, NY.

Glinsky, E. and Wainer, G. 2006. New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++. In *Proceedings of the 39th Annual Simulation Symposium*. IEEE Computer Society, Washington, DC, USA, 244–251.

Leivent, J. I. and Watro, R. J. 1993. Mathematical foundations for time warp systems. *ACM Transactions on Programming Languages and Systems 15,* 5, 771–794.

Liu, Q. 2007. Parallel Environment for DEVS and Cell-DEVS Models. *SIMULATION 83,* 6, 449–471.

Maler, O., Manna, Z., and Pnueli, A. 1992. From Timed to Hybrid Systems. In *Proceedings of the REX Workshop "Real-Time: Theory in Practice", volume 600 of Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 447–484.

Manna, Z. and Pneuli, A. 1993. Verifying Hybrid Systems. In *Hybrid Systems, volume 736 of Lecture Notes in Computer Science*. Springer, Berlin / Heidelberg, 4–35.

Martin, D. E., McBrayer, T. J., Radhakrishnan, R., and Wilsey, P. A. 1999. WARPED: A TimeWarp Parallel Discrete Event Simulator (Documentation for version 1.0). Available online at http://www.ececs.uc.edu/ paw/warped/doc/index.html, accessed December 2007.

Mattern, F. 1993. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18,* 4, 423–434.

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

Nutaro, J. 2003. Parallel Discrete Event Simulation with Application to Continuous Systems Simulation. Ph.D. thesis, University of Arizona, Department of Electrical and Computer Engineering.

Nutaro, J. and Sarjoughian, H. 2004. Design of Distributed Simulation Environments: A Unified System-Theoretic and Logical Processes Approach. *SIMULATION: Transactions of The Society for Modeling and Simulation International 80,* 11, 577–589.

Rajan, R. and Wilsey, P. A. 1995. Dynamically switching between lazy and aggressive cancellation in a time warp parallel simulator. In *SS '95: Proceedings of the 28th Annual Simulation Symposium*. IEEE Computer Society, Washington, DC, USA, 22–30.

Rönngren, R. and Liljenstam, M. 1999. On event ordering in parallel discrete event simulation. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*. IEEE Computer Society, Washington, DC, USA, 38–45.

Soliman, H. M. 1999. On the selection of the state saving strategy in time warp parallel simulations. *Transactions of the Society for Computer Simulation International 16,* 1, 32–36.

Wieland, F. 1997. The threshold of event simultaneity. In *PADS '97: Proceedings of the eleventh workshop on Parallel and distributed simulation*. IEEE Computer Society, Washington, DC, USA, 56–59.

Xu, Q. and Tropper, C. 2005. XTW, A Parallel and Distributed Logic Simulator. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, Washington, DC, USA, 181–188.

Young, C. H., Abu-Ghazaleh, N. B., and Wilsey, P. A. 1998. OFC: A Distributed Fossil-Collection Algorithm for Time-Warp. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*. Springer-Verlag, London, UK, 408–418.

Zeigler, B. P., Ball, G., Cho, H., Lee, J., and Sarjoughian, H. 1999. Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions. In *1999 Fall Simulation Interoperability Workshop*.

Zeigler, B. P., Praehofer, H., and Kim, T. G. 2000. *Theory of Modeling and Simulation: Second Edition*. Academic Press, San Diego, CA.

Zeng, Y., Cai, W., and Turner, S. J. 2004. Batch based cancellation: a rollback optimal cancellation scheme in time warp simulations. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*. ACM, New York, NY, USA, 78–86.