A Unified View of Time and Causality and its Application to Distributed Simulation

James J. Nutaro
Arizona Center for Integrative Modeling and Simulation
Electrical and Computer Engineering Department
University of Arizona, Tucson, AZ 85721-0104
Email: nutaro@ece.arizona.edu

Hessam S. Sarjoughian
Arizona Center for Integrative Modeling and Simulation
Computer Science and Engineering Department
Arizona State University, Tempe, AZ 85287-5406
Email: sarjoughian@asu.edu

Keywords: Causality, DEVS, Distributed, Logical Processes, Simulation, System Theory, Time.

Abstract
This paper presents a framework for distributed simulation that is based on system-theoretic and logical-process concepts. The framework describes a three-part world-view for developing simulation models. These are modeling formalisms, abstract simulators, and computational environments. A unified view of time and causality allows for the application of system-theoretic notions of causality within a distributed simulation environment. Within this framework, we introduce a unified notion of causality. Further we describe an approach for developing distributed simulation models which evolve from modeling constructs to simulation algorithms and their implementations. The framework is exemplified using the Discrete EVent System Specification (DEVS) modeling formalism, its abstract simulator, and a parallel algorithm that implements the abstract simulator.

INTRODUCTION

The design and verification of distributed algorithms frequently relies on an event-oriented model of a process in which processes communicate via messages (e.g., [1]). Given a set of processes, a single process is described as a sequence of states and events. A process changes state in response to events that can be generated internally or that arrive from some other process. Formally, such a set of interacting processes can be described by a decomposed, partially ordered set (deposet, see, for example, [2]). This formalization is useful for proving that particular algorithms have desired characteristics such as safety and liveness.

An extension of this formalism, in which messages are assigned time stamps, describes the logical-process approach to distributed discrete event simulation [3]. The local causality constraint restricts the allowable sequences of local states and events in such a way that the global sequence of states and events is causally consistent.

While event oriented models can be simulated by logical process based algorithms, discrete event systems have been described which can not be easily or straightforwardly mapped into an event oriented modeling paradigm [4]. Examples of such models are partial differential equations and systems of ordinary differential equations approximated by quantized state systems [5]. One approach to this problem is to handle each model or particular types of models on a case by case basis, e.g., the discrete time algorithm presented by [6]. Another, more generally applicable, approach is to map system-theoretic formalisms into a logical process world-view.

This paper presents a portion of the abstract time systems theory (see [11]) and the discrete event system specification formalism (see [5],[7-8]). The problems of time and causality in distributed simulation systems are revisited in the context of simulating causal time systems. A model of a simulation process is presented that is sufficiently general to represent, in a computationally friendly form, the class of discrete (time and event) causal systems. The usefulness of this model is shown in the development of a simple parallel algorithm for simulating DEVS models. A sketch of the correctness proof for the algorithm is also presented.

SYSTEMS, CAUSALITY, AND CAUSAL PRECEDENCE

An I/O relational system is a system for which the input quantities and output quantities can be distinguished [11][5]. A relational system S is described as the cross product of two sets X and Y denoting the input set and output set respectively. The system S is an I/O functional system if S is a function, assuming some particular initial state if necessary. At this level of abstraction, the system is a collection of inputs to the system and the resulting outputs. For example, the function {(T,F),(F,T)} describes a system that behaves as a logical not gate.

An abstract time system is a system with input and output objects that are functions of an independent variable.

These types of input and output objects are called abstract time functions. An abstract time function is a map from a time base T to a set of values. A set T can be used as an abstract time base so long as a total order, denoted by <, and an equivalence relation, denoted by =, is defined for T such that if $t_1 = t_2$ then it is not the case that $t_1 < t_2$. The real numbers under the usual < and = relations is an example of such a time base.

An abstract time system is causal if the system output is a function of past and current, but not future, inputs. That is, the future cannot effect the past. In fact, there exist two notions of causality [11]. The first requires that the system outputs be a function of past and current inputs. Such a system is weakly causal, or simply causal. The second requires that the system outputs be a function of past inputs only. Such a system is called strongly causal.

Logical Processes

Consider a system described by a logical process [6][12]. The logical process is described by an initial state and a function that takes a sequence of messages (or shared variable values) over a simulated time interval from t_0 to t_f and produces a corresponding sequence of output messages (or shared variable values) over the same interval. Such a logical process is called *realizable* by [12]. Clearly, a realizable logical process is a weakly causal system.

A *predictable* logical process is proposed by [12] as a necessary condition for a logical process to be implemented on a parallel computer. A predictable logical process is a logical process whose output at time t is a function of the inputs received up to some time $t + \varepsilon$, where ε is a positive number. A predictable logical process is a strongly causal system.

There are two aspects of the logical process world-view that introduce problems when trying to represent general system-theoretic models. The first is the strong causality of logical processes. This presents problems when trying to simulate weakly causal systems. An example of a weakly causal system is the implicit Euler approximation of $dx/dt=f(x(t))$ by the causal (but not strongly causal, i.e. not predictable) discrete time system $x(t+h)=x(t)+hf(x(t+h))$, where h is the integration time step. The simulator for this system has two computational loops. The outer loop advances the simulation clock. The inner loop solves the fixed-point problem for a particular time step. Similar systems can be represented by weakly causal DEVS models (see [13]). Non-predictable logical processes are considered by [6], but they are not addressed in a systematic way.

The second difficulty is apparent from a careful study of the process model from which the logical process model is derived. The input streams are sequences of messages with exactly one message being processed at each point in time. How do we manage simultaneous events arriving from multiple sources? This problem has appeared in the parallel discrete event simulation literature as the simultaneous event problem (see, for example, [14-15]). The solution has almost invariably relied on further information about the model itself (e.g. priorities), or was solved by fiat (i.e. first come, first served).

For example, suppose we wanted to model an n-body gravitational problem using a collection of n logical processes, one for each body. A body is a system with an n dimensional input vector where each element of the vector is provided by a distinct logical process. It is clear that each element of the input vector acts on the system simultaneously.

In a logical process world-view, a simulation protocol could be built into the models themselves that control how the system as a whole advances forward in time. For instance, each logical process might wait to receive a message from all other logical processes before moving onto the next time step. Unfortunately, we are beginning to mix the model and the simulator, causing one to be inextricably entwined with the other.

If we lose the distinction between a model and its simulator, the task of showing that simulation algorithms are correct with respect to a system-theoretic modeling formalism becomes impractical. This observation has been made by several authors (e.g. [16-17]) who have constructed correctness proofs for parallel simulation algorithms. The result of these proofs is to show that messages are processed in time stamp order. The issue of correctness with respect a modeling formalism (e.g. DEVS) is not addressed. A recent study considered separation of models and their simulators by examining the weaknesses of employing the DEVS modeling formalism or the HLA/RTI distributed simulation paradigm in isolation [10]. It discusses the benefits of unifying the system-theoretic modeling and the logical process paradigms for realization of distributed simulation models.

LOGICAL CLOCKS

Logical clocks are used for detecting causal precedence relations in a distributed computing environment. As was discussed in section 2, a causal precedence relation over a set of events and the time ordering of a set of events can not be used interchangeably when dealing with general system-theoretic models. This is a key point, since it implies that the model's notion of time, by itself, cannot be used to deduce causality. In contrast to this, the logical process world-view, with its basis in a strongly causal modeling paradigm, has generally treated time ordering and causal precedence as being interchangeable.

This observation prompts the question of how logical process simulators can be usefully employed to simulate system-theoretic models? There is substantial empirical evidence to indicate that this is the case (e.g. [18-19], and [9]). However, one cannot pick up an arbitrary logical

process simulator and say that it will be correct with respect to a particular modeling formalism.

Minimally Consistent Clocks

A minimally consistent clock describes the notion of time and its relationship to causality in general systems, where we allow the output and state at time t to be a function of inputs up to and including time t. A minimally consistent clock is one that meets the following condition [14]: Let e_1 and e_2 be two events and C a minimally consistent clock. If $e_1 \rightarrow e_2$ then $C(e_1) \leq C(e_2)$. It is easy to see that if $C(e_1) > C(e_2)$ then $\neg(e_1 \rightarrow e_2)$.

A minimally consistent clock allows two events that happened simultaneously to be causally related. When simulating weakly causal systems, the ability to discern causality between two seemingly simultaneous events is often necessary. Such a capability is needed for modeling event sequences that have a definite causal relationship, but no measurable time elapses in the course of the model interaction (see, for example, [20]).

Weakly Consistent Clocks

A weakly consistent clock meets the following condition [2][21]: Let e_1 and e_2 be two events and C be a weakly consistent clock. Then

i) If $C(e_1) > C(e_2)$ then $\neg(e_1 \rightarrow e_2)$, and
ii) If $C(e_1) = C(e_2)$ then $\neg(e_1 \rightarrow e_2)$ and $\neg(e_2 \rightarrow e_1)$.

The statement $\neg(e_1 \rightarrow e_2)$ and $\neg(e_2 \rightarrow e_1)$ is denoted $e_1 \parallel e_2$. From the definition, it can be seen that

iii) If $e_1 \rightarrow e_2$ then $C(e_1) < C(e_2)$.

In the context of weakly causal system, the model clock, when used as a weakly consistent clock, is insufficient to perform the simulation. Since several causally related function evaluations might take place at time t, we would be forced to conclude that the order in which the function evaluations take place is immaterial to the outcome.

A General Purpose Simulation Clock

A weakly consistent clock suitable for keeping time in simulations of weakly causal systems is presented by [22]. We provide a system-theoretic interpretation of the algorithm and show how it resolves the causality problem using time stamps appropriate for use in a parallel algorithm. A time stamp is a pair (t,c) where t is a model derived time-stamp (e.g. the time of next event) and c is an integer counter. The simulator maintains a time of last event (t_L,c_L) whose initial value is (0,0). When a model executes an event at *model* time t, the simulator compares that t to t_L. If $t = t_L$, then the *simulation* time of next event becomes (t,c_L+1). Note that the model time of next event is still t. If $t_L < t$, then the simulation time of next event becomes (t,0). The time of last event is then set to be the previous time of next event.

There are two rules for comparing time stamps:

1. $(t_1,c_1) < (t_2,c_2)$ if $t_1 < t_2$ or $t_1 = t_2$ and $c_1 < c_2$, and
2. $(t_1,c_1) = (t_2,c_2)$ if $t_1 = t_2$ and $c_1 = c_2$.

The operation of the clock can be imagined by drawing a directed, a-cyclic graph where each node represents a model that undergoes a zero time state transition (i.e., processes a zero time event) in response to an input and each arc represents an input to the model. If cycles are present in the graph, they are broken by inserting nodes to represent a particular model at different instances in time (i.e., one node per model state change is allowed in the graph). The input-free nodes are labeled 0. These are the first nodes to execute events at, say, time t. The neighbors of these nodes are labeled 1. They are the next set of nodes to execute events at time t. Similarly, their neighbors are labeled 2. Eventually, this procedure will label every node in the graph. Sets of nodes with identical labels can be computed in any order. Otherwise, nodes must be computed in label order. The labels are the numbers assigned to different zero time events by the second field of the simulation clock.

THE DEVS FORMALISM

The DEVS formalism uses two types of models to describe a system. Atomic models represent non-decomposable processes, entities, or other types of system. More complex models are constructed hierarchically. A coupled model is a multi-component model constructed from atomic models and other coupled models.

An atomic model is described by a state set, output set, and input set and a collection of functions that define its dynamic behavior. The internal transition function describes the autonomous behavior of the system. The external transition function describes the input response of the system. The confluent transition function is used to resolve the case where an internal and an external transition occur simultaneously. The time advance function indicates the time that must elapse before the next internal event occurs, assuming that no input becomes available in the interim. The output function gives the output of the system as a function of its state immediately before the internal transition function.

Formally, an atomic model is represented by a structure $< S, X, Y, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$ where

S is the set of system states
X is the set of input events
Y is the set of output events
δ_{int}, : $S \rightarrow S$ is the internal transition function,
$\delta_{ext} : Q \times X^b \rightarrow S$ is the external transition function,
where $Q = \{ (s,e)$ such that $s \in S$ and $0 \leq e \leq ta(s) \}$,

δ_{conf} : S × X^b → S is the confluent transition function,
ta : S → $\mathbf{R}_+ \cup \{\infty\}$ is the time advance function, and
λ : S → Y^b is the output function.

Note that the formalism allows bags of input and output events to be consumed and produced, respectively, by the system.

A coupled model is specified in terms of a set of atomic components and a coupling specification. The closure under coupling property for coupled models (see [5]) ensures that a coupled model can be reduced to an atomic model that is equivalent at the I/O functional level. This allows for the inclusion of coupled models as components of other coupled models and so enables hierarchical model construction.

Formally, a coupled model is described by a structure N = <X,Y,M,Z> where X and Y are the model input and output sets, M is a set of component models subject to the constraint the N ∉M (i.e., the network model can not be a component of itself), and Z is a coupling specification. The coupling specification is a set of functions that describe three types of mappings. The functions z_{Nm}, where m ∈M are the external input couplings. These map the coupled model input set X to the component model input sets X_m. Similarly, the functions z_{mN} are the external output couplings that map the component output sets Y_m to the network model's output set Y. Finally, internal couplings are described by the functions z_{mk} where m,k ∈M and z_{mk} maps the output set Y_m to the input set X_k.

For the purposes of this paper, we consider only the simulation of a flat coupled model (i.e., one in which any coupled components have been reduced to their atomic equivalents). Given an input trajectory, the resulting state and output trajectories are computed as follows. Set the time of last event tL_i to zero for every component model i. Set the time of next event, call it tN, to the smallest time $tL+ta_i(s_i)$ over all models, where $ta_i(s_i)$ is the time advance of the i^{th} model, and the earliest event bag in the input trajectory, call it x.

Having computed the time of next event for this cycle, the component model input/output is determined[1]. If the time at which x is valid is tN, apply the input $z_{Ni}(x)$ to every component model i such that $z_{Ni}(x)$ is not the non-event. Also, for every component model k whose time of next event is equal to tN, compute its output function and apply the input $z_{ki}(\lambda(s_k))$ to every model i such that $z_{ki}(\lambda(s_k))$ is not the non-event. If a model received input from more than one source, the incoming events are collected into a single input bag for the model. This completes the input/output computation for the cycle.

[1]Note that message passing from one component to another is instantaneous.

Next, the model state changes are computed. For every component model whose time of next event is tN but has not received input, compute its next state using the internal transition function. For every component model whose time of next event is tN and has received input, compute its next state using the confluent transition function. For every model component whose time of next event is less than tN and has received input, compute its next state using the elapsed time value tL_i - tN and the external transition function. The time of last event for all models that changed state is then set to tN (i.e. tL_i := tN for every model i whose state has changed). This completes a cycle, and so we repeat.

COMPUTATIONAL REPRESENTATIONS OF SYSTEM FORMALISMS

A theoretical framework for studying simulation algorithms provides an abstract view of the modeling formalism under consideration. This abstraction is useful because it describes the formalism in a computationally friendly way. We focus on the I/O functional view of a model where the initial state is assumed and the input trajectory is described by an event sequence. The concatenation operator is provided to model multi-dimensional input trajectories (see section 2.1). A particular choice of clock (i.e. weakly or minimally consistent) describes the relationship between time and causality in the modeling formalism. In general, the two dimensional clock described in section 3.3 can be used to construct a weakly consistent clock for system formalisms that allow weakly causal models. However, the minimally consistent clock is not disallowed.

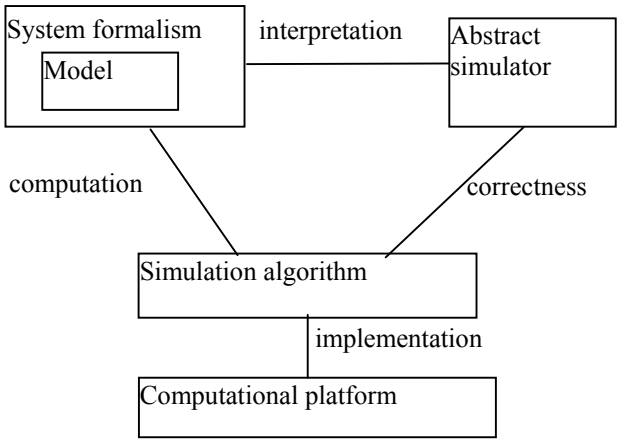

Figure 1. Elements of a modeling and simulation framework.

Figure 1 shows the organization of a simulation framework for a general modeling formalism. This view is derived from the seven-layer view presented in [10]. The models themselves are described in an algorithm neutral

way (i.e. the construction of the model description requires no knowledge of the simulation algorithm). Associated with the model is an abstract simulator that describes the rules for interpreting models described using the system formalism. This is not the same as rules for interpreting a particular model. The details of a particular model's behavior are contained within the model itself.

From the abstract simulator, a particular simulation algorithm is derived that is behaviorally identical to the abstract simulator. This is mapped onto a computational platform (e.g. hand calculator, parallel computer, cluster, or workstation) where the actual computation is performed.

The computational model that is presented in this paper fits between the system formalism and the simulation algorithm. The goal is to provide a means of expressing simulation algorithms in a form that is concise and eases the job of building correctness proofs for a computational scheme. Correctness, of course, being relative to the abstract simulator whose behavior we are trying to realize.

A computational representation of a system formalism is a structure that is, at the I/O functional level, equivalent to the formalism and whose time stamped events form of a decomposed, partially ordered set. The computational representation consists of a set of inputs values, a set of output values, and a set of time stamps associated with those values. An event is a (time,value) pair. We use the event e and its associated pair (t,v) interchangeably. We adopt the convenient notational convention $e_i = (t_i,v_i)$. We can order events by time stamp as follows

1. $e_1 \le e_2$ iff $t_1 \le t_2$
2. $e_1 < e_2$ iff $t_1 < t_2$
3. $e_1 = e_2$ iff $t_1 = t_2$

We expect time stamps to be assigned to values in a way that is consistent with our expectation that smaller time stamps indicate earlier events. If e_1 and e_2 are two causally independent events with equal time stamps, then we can concatenate their values to get a new event e_3. Concatenation will be denoted by $\bullet$. For example, let (t,a) and (t,b) be two causally independent events. Then $(t,a)\bullet(t,b) \equiv (t,\{a,b\})$. Concatenation allows bags of causally unrelated events with equal time stamps to be denoted by a single event. Concatenation is defined to be associative and commutative.

We rely on a particular interpretation of the simulation clock to determine when two events are causally independent. Most commonly, weak consistency is assumed and so $e_1 = e_2$ is taken to imply $e_1 \parallel e_2$. However, if minimal consistency is assumed, we cannot in general determine if $e_1 \parallel e_2$.

A computational representation has a state that is represented by a stack of events that record the event history of the simulation. Formally, a computational representation is described by STC = <E, G, push, eN, out> where

E is an event set
G is a set of event stacks $(e_0e_1...)$ where $e_i \in E$,
push : $G \times E \rightarrow G$ is the transition function,
eN : $G \rightarrow E$ is the next event function, and
out : $G \rightarrow E$ is the output function.

The push function is defined such that
$push((e_0e_1...e_n),e) = (e_0e_1...e_ne)$.

The functions eN and out depend on the behavior of the interpretation of the simulation clock. If we assume a weakly consistent clock, then these function are constrained such that

$eN(e_0e_1...e_n) = e$ such that $e_n < e$, and
$out(e_0e_1...e_n) = y$ such that $e_n < y \le eN(e_0e_1...e_n)$.

If a minimally consistent clock is assumed, then
$eN(e_0e_1...e_n) = e$ such that $e_n \le e$, and
$out(e_0e_1...e_n) = y$ such that $e_n \le y \le eN(e_0e_1...e_n)$.

Computational Representation of Logical Processes

We claim the above model includes the logical process model. A stack corresponds to an event history for the logical process. Given a stack s, the operation push(s,eN(s)) represents the logical process executing an event that it scheduled for itself. If an event x is received from some other the logical process, the operation push(s,x) constructs the event history that results from processing x. The out(s) function is used to determine which events should be sent to other logical processes.

A logical process simulator may or may not make use of event concatenation. For example, if a particular simulator uses a first come, first serve processing order for events with identical time stamps, concatenation is not needed. However, if events with equivalent time stamps are delivered to the logical process for sorting by the model, then concatenation provides a mechanism for doing so. Similarly, a particular logical process implementation may choose either form of clock (see for example, Yaddes which requires non-zero look-ahead for efficient simulation (see [17]) and SPEEDES which allows for it (see [23]).

Computational Representation of DEVS Atomic Models

We consider only atomic DEVS models. Let s be the stack that represents the event history of the system up to some time t. The next event is determined by one of three operations. An internal event is represented by the operation push(s,eN(s)). Given an input event x, the external transition function is represented by push(s,x). The

confluent transition function is modeled by the operation push(s,eN(s)•x). The output function is modeled by out(s). The time advance function is wrapped up in the time stamp of the eN(s) event, as will be shown in the next section. For a more formal treatment of the relationship between DEVS and simulation processes, see [24].

It is clear from the preceding discussion that a logical process simulator is not necessarily correct with respect to the DEVS formalism. Such a simulator might violate the formalism in several ways. For example, selection of a minimally consistent clock that uses model derived time stamps will not produce results that are correct with respect to DEVS [4]. Similarly, the algorithm must be sensitive to how preemption of internal events is handled in order to maintain the semantics of the time advance function. Lastly, DEVS specifies event bags and the confluent transition function for managing simultaneous events. These features might not be supported by a logical process simulator.

Correctness of a Parallel Algorithm

In this section, we develop a simple parallel simulation algorithm in terms of the above model. The algorithm is a basic, event stepped algorithm equivalent to a single level Parallel DEVS abstract coordinator (see [5]). We show that the algorithm is correct by hypothesizing a correct final stack configuration and demonstrating that the algorithm constructs that stack.

Let $<E_i, G_i, push_i, eN_i, out_i>$ denote the i^{th} simulator and assume we have N of them labeled 1,2,...,N. We use Φ to denote no event (i.e. x•Φ=Φ). It is assumed that the clock is weakly consistent. The `for each` blocks are executed in parallel.

Algorithm 1: Event stepped simulation

```
1  tN := min{out_i(stc_i).t} for all i in [1,N]
2  while (tN < t_stop)
3        for each i in [1,N]
4              if (eN_i(stc_i).t = tN)
5                    x_i := eN_i(stc_i)
6              else
7                    x_i := Φ
8              end
9        end
10       for each i in [1,N] such that out_i(stc_i).t = tN
11             for each j in [1,N] such that i influences j
12                   x_j := x_j • out_i(stc_i)
13             end
14       end
15       for each i in [1,N] such that x_i ≠ Φ
16             stc_i := push(stc_i,x_i)
17       end
18       tN := min(out_i(stc_i).t) for all i in [1,N]
19 end
```

We sketch a proof for the restricted case of two simulators, call them 1 and 2. Suppose the end result of a run should be 1 = $(x_1x_2x_3...x_n)$ and 2 = $(z_1z_2...z_m)$. The simulator is trivially correct if 1 and 2 are empty stacks (i.e. there are no events to process). Suppose its correct up to $(x_1x_2x_3...x_p)$ and $(z_1z_2...z_q)$ where $p < n$ and $q \le m$. Without loss of generality, we consider how the stack $(x_1x_2x_3...x_{p+1})$ is constructed. A new event can be added as a result of an internal transition, external transition, or a confluent transition.

Assume the next simulation event for 1 is an internal transition. That is, $x_{p+1}=eN_1(x_1x_2x_3...x_p)$ and $out_1(x_1x_2x_3...x_p).t=x_{p+1}.t$ and $x_{p+1}.t < out_2(z_1z_2...z_q).t$. In this case, 1 processes some internal event and generates a (possibly null valued) output event, but does not receive input from 2. Line 18 ensures that $tN=out_1(x_1x_2x_3...x_p).t$. By hypothesis then, $tN=eN_1(x_1x_2x_3...x_p).t$. So line 4 evaluates to true and at line 5, $x_1=eN_1(x_1x_2x_3...x_p)$. Since $tN<out_2(z_1z_2...z_q).t$, line 12 is not executed. Finally, line 15 evaluates to true and at line 16 we push x_1 onto the stack. Since x_1 is x_{p+1}, we get the desired result.

Now, assume the next simulation event for 1 is an external transition. That is, $x_{p+1}.t=out_2(z_1z_2...z_q).t$ and $eN_1(x_1x_2x_3...x_p)<out_2(z_1z_2...z_q)$. In this case, 1 processes some external event that is received from 2. Line 18 ensures that $tN=out_2(z_1z_2...z_q)$. So line 7 is executed setting $x_1 = \Phi$. Line 10 evaluates to true for 2 and, since 2 influences 1, we end up with $x_1=out_2(z_1z_2...z_q)$. Again, line 15 evaluates to true and at line 16 we push x_1 onto the stack. Since x_1 is x_{p+1}, we get the desired result.

Finally, assume the next simulation event for 1 is a confluent transition. That is, $x_{p+1}= out_2(z_1z_2...z_q)•eN_1(x_1x_2x_3...x_p)$ and $eN_1(x_1x_2x_3...x_p).t =out_1(x_1x_2x_3...x_p).t=x_{p+1}.t$. In this case, 1 processes a bag of events constructed from an external event received from 2 and an internal event. Line 18 ensures that $tN=out_2(z_1z_2...z_q).t=out_1(x_1x_2x_3...x_p).t$. By hypothesis, $tN=eN_1(x_1x_2x_3...x_p).t$. So line 4 evaluates to true, setting $x_1=eN_1(x_1x_2x_3...x_p)$. Line 10 evaluates to true for 2 and, since 2 influences 1, we end up with $x_1=out_2(z_1z_2...z_q)•eN_1(x_1x_2x_3...x_p)$. Finally, line 15 evaluates to true and at line 16 we push x_1 onto the stack. Since x_1 is x_{p+1}, we get the desired result.

Additional properties that we would like to show are safety (i.e. the algorithm only applies internal, external, or confluent events to the simulators), liveness (at least one event is processed per pass), and correct termination (the simulation halts when the desired stacks have been created). Briefly, safety holds since any case other than an internal, external, or confluent transition causes line 15 to evaluate to false. Liveness holds because the simulation clock is always advanced to the time of the next simulation event. The algorithm terminates when the next simulation event has a time stamp greater than t_{stop}.

CONCLUSIONS

The model presented in this paper provides a unified approach for describing time and causality in parallel simulation protocols when those protocols are required to be correct with respect to system-theoretic modeling principles. The model provides a unique capability to describe algorithms that can simulate models with a DEVS representation at the I/O functional level. The model maintains important aspects of the logical process model that has been useful for studying distributed and parallel simulation algorithms.

ACKNOWLEDGEMENT

This research was partially supported by NSF Next Generation Software (Grant No. EIA-9975050) and Scaleable Enterprise System (Grant No. DMI-0122227) programs.

REFERENCES

[1] Fujimoto, R.M., *Parallel and Distributed Simulation Systems*, John Wiley and Sons, Inc., 2000.

[2] Garg, V.K., *Principles of Distributed Systems*, Kluwer Academic Publishers, Boston, 1996.

[3] Reed, D.A., R.M. Fujimoto, "Mutlicomputer Networks: Message-based parallel processing", MIT Press, Cambridge, Mass., 1987.

[4] Zeigler, B.P., G. Ball, H. Cho, J.S. Lee, H.S. Sarjoughian, "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions", SIW, March 1999.

[5] Zeigler, B.P., H. Praehofer, T.G. Kim, *Theory of Modeling and Simulation, 2nd edition*, Academic Press, New York, 2000.

[6] Bagrodia, R., "A Unifying Framework for Distributed Simulation", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 4, pp. 248-385, 1991.

[7] Wymore, W.A., *Model-based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*, CRC, Boca Raton, 1993.

[8] Sato, R., "Realization Theory of Discrete-event Systems and its Application to the Uniqueness and Universality of DEVS", *Int. J. Of General Systems*, Vol. 30, No. 5, pp. 513-549, 2001.

[9] Wainer, G., "Improved Cellular Models with Parallel Cell-DEVS", *TRANSACTIONS of the Society for Computer Simulation*, Vol. 17, No. 1, 2000.

[10] Sarjoughian, H.S., B.P. Zeigler, "DEVS and HLA: Complementary Paradigms for M&S?", *TRANSACTIONS of the Society for Computer Simulation*, Vol. 17, No. 4, pp. 187-197, 2000.

[11] Mesarovic, M.D., Y. Takahara, *Abstract Systems Theory*, Springer-Verlag, New York, 1989.

[12] Misra, J., "Distributed Discrete Event Simulation", *Computing Surveys*, Vol. 18, No. 1, pp. 39-65, 1986.

[13] Kofman, E., "Quantization--Based Simulation of Differential Algebraic Equation Systems", Technical Report LSD0203, LSD, Universidad Nacional de Rosario, 2002.

[14] Rönngren, R., M. Liljenstam, "On Event Ordering in Parallel Discrete Event Simulation", Thirteenth Workshop on Parallel and Distributed Simulation, pp 38-45, 1996.

[15] Wieland, F., "The Threshold of Event Simultaneity", *Transactions of the Society for Computer Simulation International*, Vol. 16, No. 1, 1999.

[16] Frey, P., R. Radhakrishnan, H.W. Carter, P.A. Wilsey, P. Alexander, "A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation", *IEEE Trans. on Software Engineering*, Vol. 28, No. 1, pp. 58 - 78, 2002.

[17] Ghosh, S., "On the Proof of Correctness of Yet Another Asynchronous Distributed Discrete Event Simulation Algorithm (YADDES)", *IEEE Trans. on Systems, Man, and Cybernetics-Part A: Systems and Humans*, Vol. 26, No. 1, pp. 68-80, 1996.

[18] Prähofer, H. and G. Reisinger. "Distributed Simulation of DEVS-based Multiformalism Models", In AIS '94. Gainesville, FL, December 1994.

[19] Liao C., A. Motaabbed, D. Kim, B.P. Zeigler, "Distributed Simulation of Sparse Output DEVS", Proceedings of AI, Simulation, and Planning in High Autonomy Systems, Tucson, AZ, September, IEEE/CS 1993.

[20] Rieffel, E., "An Introduction to Quantum Computing for Non-Physicists", *ACM Computing Surveys*, Vol. 32, No. 3, pp. 300-335, 2000.

[21] Schwarz, R., F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", *Distributed Computing*, Vol. 7, No. 3, pp. 149-174, 1994.

[22] Jha, V., R. Bagrodia, "A Unified Framework for Conservative and Optimistic Distributed Simulation", 8th Workshop on Parallel and Distributed Simulation, pp. 12-19, 1994.

[23] Steinman, J.S., "SPEEDES: A multiple-synchronization environment for parallel discrete event simulation", *The International Journal for Computer Simulation*, Vol. 2, No. 3, pp. 251-286, 1992

[24] Nutaro, J.J., "Time Management and Interoperability in Distributed Discrete Event Simulation", Master's Thesis, University of Arizona, Department Electrical and Computer Engineering, 2000.