PARALLEL DISCRETE EVENT SIMULATION WITH APPLICATION TO CONTINUOUS SYSTEMS

by

James Joseph Nutaro

Copyright © James Joseph Nutaro 2003

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2003

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: _____

ACKNOWLEDGEMENTS

I would like to thank my advisor, Bernard Ziegler, and Hessam Sarjoughian, at the University of Arizona, for years of mentoring and encouragement.

Thank you Angela for your patience and support.

I would like to also thank Phil Hammonds, at the USGS, for providing access to the Beowulf cluster on which the software for this dissertation was developed.

TABLE OF CONTENTS

LIST OF FIGURES....7
LIST OF TABLES....9
ABSTRACT....10
1. INTRODUCTION....12
2. SIMULATION ALGORITHMS FOR DISCRETE EVENT SYSTEMS....16
2.1. Discrete Event Input/Output Functions....18
2.2. Atomic Models....20
2.3. Coupled Models....31
3. ALGORITHMS FOR PARALLEL DISCRETE EVENT SIMULATION....39
3.1. A Model for Distributed Computing Systems....39
3.2. Logical Clocks....41
3.3. Consistent Cuts....44
3.4. Computing Minimum Values....47
3.5. Global Virtual Time....50
4. CAUSALITY AND CAUSAL PRECEDENCE....53
4.1. Systems, Causality, and Causal Precedence....54
4.2. An I/O Functional Model of a Distributed Computing System....57
4.3. Causal Precedence in Multicomponent Systems....64
4.4. Logical Processes....66
4.5. Logical Clocks in Distributed Simulation....69
5. A FRAMEWORK FOR PARALLEL SIMULATION OF DEVS MODELS....72
5.1. A Framework for Distributed Simulation....72
5.2. Computational Representation of Logical Processes....76
5.3. Computational Representation of DEVS Atomic Models....77
5.4. An Event Stepped Parallel Algorithm....78
6. OPTIMISTIC SIMULATION OF DEVS MODELS....82
6.1. A Time Warp Algorithm for DEVS Models....84
6.2. Checkpointing Theorem....90
6.3. Causality Theorems....92
6.4. Retention Theorems....95
6.5. Sequencing Theorem....97
6.6. Liveness Theorems....98
6.7. Correct Simulation of DEVS Models....101
7. RISK-FREE OPTIMISTIC SIMULATION OF DEVS MODELS....103
7.1. A Risk Free Optimistic Algorithm....105
7.2. Detecting the End of a Parallel Segment....108
8. DEVS APPROXIMATIONS OF CONTINUOUS SYSTEMS....113
8.1. Concepts....114
8.2. Adams-Bashforth Type Quantized State Systems....117

TABLE OF CONTENTS - CONTINUED

8.3. Local Truncation Error of the DEVS-Adams-Bashforth Methods....120
8.4. Convergence of the DEVS-Adams-Bashforth Methods....122
8.5. Linear Stability for DAB1 and DAB2....127
8.6. Vector Valued States....129
8.7. Partial Differential Equations....131
8.8. Application of DAB1 to Finite Difference Schemes....132
9. THE SOD SHOCK TUBE PROBLEM....143
9.1. Activity Tracking Property of the DEVS-Based Numerical Scheme....146
9.2. Parallel Simulation....153
10. THE WAVE EQUATION IN ONE DIMENSION....156
10.1. Digital Wave Guide Networks....156
10.2. Solving the Wave Equation on a Parallel Computer....161
11. CONCLUSIONS....172
REFERENCES....176

LIST OF FIGURES

Figure 2.1. State trajectory (S) and output trajectory (Y) for the counter model.....26
Figure 2.2. Valid couplings between components.....32
Figure 3.1. A space-time diagram for a two processor distributed system.....40
Figure 3.2. A consistent cut (3.2a on left) and an inconsistent cut (3.2b on right).....44
Figure 3.3. A consistent cut that accounts for all messages crossing the cut line.....45
Figure 3.4. A binary tree used to compute global minimums on a parallel computer.....49
Figure 4.1. A multicomponent system consisting of several processes and a message passing system.....62
Figure 5.1. The elements of a modeling and simulation framework.....73
Figure 6.1. An output segment for a discrete event model.....83
Figure 6.2. A road map for the Time Warp algorithm correctness proof.....89
Figure 7.1. Thrashing in time warp when two processes have similar time advance values.....104
Figure 7.2. Process A waits for process B to find its next (incorrect) output. The black squares are output events and the white squares represent internal events without associated outputs.....109
Figure 8.1. Solution to (solid line) and its approximate solution by explicit Euler (dotted line).....114
Figure 8.2. Solution to (solid line) and its approximate solution by the quantized integration scheme (8-6) and (8-7).....116
Figure 8.3a. The absolute error in the DAB1 approximation to with y(0) = 1 as a function of the integration quantum.....124
Figure 8.3b. The absolute error in the DAB2 approximation to with y(0) = 1 as a function of the integration quantum.....124
Figure 8.4. Oscillations in the solution of dy/dt = -y with y(0) = 1 and D = 0.3 using DAB1.....128
Figure 8.5. Oscillations in the solution of dy/dt = -y with y(0) = 1 and D = 0.3 using DAB2.....129
Figure 8.6. Movement of a solution within a quantum volume for a two-dimensional system.....131
Figure 8.7. An illustration of the effect of the integration quantum on near steady state behavior for the heat equation in one dimension.....140
Figure 8.8. The 18 possible arrangements of a center cell and its neighboring cells.....141
Figure 9.1. A DEVS coupled model that approximates the conservation law. The symbol $\int$ denotes the quantized integrator, $\sum$ is the summer, and Φ is the flux model.....144
Figure 9.2. Computed wave fronts for the Sod shock tube problem.....148
Figure 9.3. Number of internal state changes required to compute the wave fronts depicted in Figure 9.2.....148

LIST OF FIGURES - CONTINUED

Figure 9.4. Difference between the final and initial state of the shock tube for a 0.2 second simulation using 200 cells.....151
Figure 10.1. A wave encountering encountering a discontinuity in the medium.....160
Figure 10.2. A homogenous (below) and inhomogeneous (above) string.....162
Figure 10.3. Solution to equation (10-1) solved on a homogenous string (above) and inhomogeneous string (below).....163
Figure 10.4. Execution times for the homogenous string problem with 50k cells.....164
Figure 10.5. Execution times for the inhomogeneous string problem with 50k cells.....165
Figure 10.6. Speedup for the homogenous string problem with 50k cells.....165
Figure 10.7. Speedup for the inhomogeneous string problem with 50k cells.....166
Figure 10.8. Solution to (10-1) using a narrow initial pulse with a string whose left two thirds and right one third have densities of 10 and 1, respectively.....169

LIST OF TABLES

Table 9.1. Solution parameters and speedup for DEVS and discrete time (DTSS) techniques.....149
Table 9.2. Anticipated speedup of the discrete event simulation with respect to the discrete time simulation.....152
Table 9.3. Corrected anticipated speedup of the discrete event simulation with respect to the discrete time simulation.....153
Table 9.4. Speedup of the parallel discrete event and discrete time algorithms.....156
Table 10.1. Execution time and speedup for the discrete event solution using narrow initial conditions.....168
Table 10.2. Execution time and speedup for the discrete event solution using narrow initial conditions and one less region of differing density.....169
Table 10.3. Execution time for the discrete time and discrete event simulation of the short and long string problems as a function of the number of processors used in the simulation. 170
Table 10.4. Relative speedup of the discrete event scheme with respect to the discrete time scheme as a function of the number of processors used to execute the simulation. 172

ABSTRACT

Recent advances in discrete event modeling of continuous systems have emphasized the need for high performance simulation engines. This need is particularly acute when discrete event methods are applied to the numerical solution of partial differential equations. Accurate approximations can require thousands, or even millions, of cells. The corresponding requirements for memory and computing power can readily exceed what is available on a single processor computer.

Discrete event simulations are characterized by asynchronous and irregular, random, or data dependent behavior. This makes parallel algorithm design particularly challenging. Known parallel discrete event simulation algorithms have been developed in terms of event and process oriented world views. In contrast to this, the Discrete Event System Specification (DEVS) forms the foundation of research into discrete event approximations of continuous systems.

While event and process oriented models can be expressed in terms of the DEVS modeling formalism, there are DEVS models that do not seem to have an equivalent representation in the event or process oriented world views. This leaves open the question of how existing parallel discrete event simulation algorithms must be adapted in order to simulate DEVS models.

In this dissertation, discrete simulation algorithms are built up from the basic definition of a discrete event system. The parallel algorithms that are developed through this approach are shown to operate correctly. To conclude this study, these algorithms

are applied to producing numerical solutions of a hyperbolic conservation law (Sod's shock tube problem) and the wave equation.

1. INTRODUCTION

This dissertation is concerned with the field of parallel discrete event simulation, and in particular with its application to solving very large systems of equations. The latter statement, which implies discrete event approximations of continuous systems, is as yet an untouched application of parallel discrete event simulation. While parallel discrete event simulation has been an active area of research for nearly thirty years, researchers have focussed almost exclusively on fast execution of process and event oriented models of discrete event systems. This is problematic when trying to apply parallel discrete event simulation to models that can not be expressed in terms of a process, or event oriented, world view. In particular, discrete event simulation of continuous systems that are approximated by DEVS models can not be readily accomplished in these world views.

The need to develop new parallel discrete event simulation algorithms is apparent for two reasons. The first is that parallel discrete event simulation algorithms developed for process and event oriented models can not correctly simulate DEVS models. While the development of new algorithms is greatly simplified by building on existing algorithms, the correctness of any new algorithm must be established from basic principles.

The second reason for developing new algorithms is an interest in new applications. Continuous systems can exhibit behavior that is quite different from the types of applications that are the domain of most existing parallel discrete event

simulation systems. One of the more striking differences is the prevalence of short, often instantaneous, feedback loops.

The first part of this dissertation is concerned with identifying a minimal set of properties that are necessary for a DEVS simulation algorithm to be correct. The rules for simulating DEVS models are precisely specified as a part of a formal approach to systems modeling. These rules are studied in Chapter 2 and a method of proof is developed for demonstrating that a simulation algorithm adheres to these rules.

Discrete event simulation in the DEVS world view incorporates well-defined concepts of causality and simultaneity that appear in general systems theory. Similar, but distinct, notions of time, causality, and simultaneity have been developed in the context of distributed algorithms. It is this latter view that has taken hold in parallel algorithms for process and event oriented models. Some necessary background in parallel computing is presented in Chapter 3 prior to developing a unified theory of time and causality in Chapter 4. A framework for developing parallel simulation algorithms is presented in Chapter 5.

At this point, we have in hand sufficient material to develop two new algorithms. In Chapter 6 a variant of the Time Warp algorithm is developed and a proof of its correctness is constructed. In Chapter 7, some of the short comings of this algorithm are discussed in the context of simulating continuous systems. A variant of the Breathing Time Bucket algorithm is developed to address these shortcomings. As before, a proof of the algorithm's correctness is developed.

The algorithm developed in Chapter 7 is applied to the Sod shock tube problem and solutions of the wave equation in one dimension. In order to solve the Sod shock tube problem it is necessary to introduce some elements of the theory of quantized integrators. An introduction to quantized integrators is provided in Chapter 8. The introduction is extended somewhat in order to discuss the use of quantized integrators in very large, sparse systems.

In Chapter 9, the Sod shock tube problem is solved using both parallel discrete event and parallel discrete time algorithms. The solution of this problem demonstrates the effectiveness of the parallel discrete event simulation relative to the parallel discrete time simulation. The relative speedup of the discrete event scheme can be roughly anticipated by an activity argument that is presented after the experimental results have been obtained. This argument will suggest that, for a given error tolerance, the discrete event scheme will always have an advantage, in terms of time to solution, over its discrete time counterpart.

The one dimensional wave equation can be solved using digital wave guide techniques that, in many cases, have both discrete time and discrete event implementations. Here we are able to get an identical solution using both discrete time and discrete event simulation algorithms and so a comparison of computational effort provides a good measure of relative performance. The digital wave guide method is briefly introduced in Chapter 10. The execution times of a parallel discrete time and parallel discrete event implementation of the technique are compared in order to demonstrate the relative advantage of the discrete event approach. This experiment

highlights the advantage of using a discrete event scheme when a system exhibits localized activity.

2. SIMULATION ALGORITHMS FOR DISCRETE EVENT SYSTEMS

The problem of simulating a discrete event system, in the context of this study, is the following: given a state transition or coupled component representation of a discrete event system (which will be defined shortly), and an appropriate input trajectory, compute the resulting output trajectory. This simulation problem can be seen as one of finding computational schemes for reducing models at a lower level of abstraction (i.e., described as state transition or coupled component models) to a more abstract input/output (I/O) function (see [Zeigler 2000],[Klir 1969]). We will take advantage of this point of view when constructing parallel algorithms for simulating discrete event systems.

For readers unfamiliar with discrete event systems, an analogy in the continuous world may be helpful. Suppose we have a linear system (this being an example of a coupled component model) described by

$$2\text{-}1) \qquad \begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx \end{aligned} \quad .$$

The corresponding I/O functional representation of this system is given by its transfer function

$$2\text{-}2) \quad Y(s) = C(sI - A)^{-1} x_0 + C(sI - A)^{-1} B\, U(s) \quad .$$

Simulating this system is unnecessary, since the system output at any point is determined by evaluating (2-2). It can be seen that the transfer function is a map from the initial state x_0 and an input trajectory U to an output trajectory Y.

Suppose, however, that we could not reasonably construct (2-2) either because the linear system is too large, or we have, instead, a non-linear system. In this case, we must resort to calculating the system behavior directly. When dealing with discrete time systems, this calculation is relatively straight forward. For continuous systems, we could transform the continuous system into some discrete time approximation. Computation using the corresponding discrete time system can then be done on a computer.

For discrete event systems, two things are needed for simulation. The first necessary item is an unambiguous description of the discrete event system. A method for describing discrete event systems is provided by the Discrete Event System Specification (DEVS). Secondly, a framework is needed in which it is possible to reason about the correctness of parallel discrete event simulation algorithms. While the latter is less significant when constructing parallel algorithms for discrete time systems, the complexity of parallel discrete event simulation algorithms requires a more rigorous approach.

The DEVS formalism describes discrete event systems in terms of hierarchical, modular structures [Zeigler 2000]. The precise mathematical structures used to describe discrete event systems provide a foundation for constructing simulators that are provably correct. The mathematical description of a DEVS model is entirely separate from the simulator that is used to explore its dynamic behavior. Any correct simulator will produce exactly the same dynamic behavior for a given model.

As it happens, there are many correct simulators for DEVS models. One such simulator is presented here and some key properties of that simulator are picked out to

provide compact correctness criteria for use in designing new simulators. Not only will this be helpful when designing new simulation algorithms, it will also aid in constructing correctness proofs for the parallel simulation algorithms that are the focus of this study.

2.1. Discrete Event Input/Output Functions

When a system is viewed from the 'outside' it is only possible to observe things that act on the system externally (i.e., inputs) and the external response of the system to those effects (i.e., outputs). If we have some information about the internal state of the system prior to its being acted upon, it is possible to uniquely determine the response of the system in response to a particular stimulus (see, e.g., [Klir 1969], [Mesarovic 1989], [Zeigler 2000]).

For example, suppose we have a candy machine that is stocked with red licorice and black licorice. The machine is stocked in such a way that if the last customer received red licorice, then the next customer will receive black licorice and vice versa. We could think of this candy machine as having two states, red and black, that describe what the next customer will receive (more precisely, the output function is the identity function and so we can easily infer the machine state from its output).

If we purchase candy from the machine without knowledge of what the last customer received (i.e., without knowing the state of the machine), we have no way of knowing whether we will receive red or black licorice. However, if we wait long enough for some other customer to purchase candy, then we can determine whether our purchase

(assuming it is the next purchase) will produce red or black licorice. In fact, we can determine what kind of candy any customer will receive from this point on.

Suppose we sit for a day and record the time at which each customer purchases candy and the type of licorice that the customer receives. If we plot these events on the time axis of two graphs, with purchases on the input graph and ejected licorice on the output graph, then we have a discrete event input/output function that describes one behavior of the candy machine (i.e., the behavior observed that day). The primary characteristic of the function (in fact, of any discrete event input output function) is that it is constructed from piecewise constant segments of arbitrary (possibly zero) length.

More generally, we can think of an I/O functional system as being a function that maps input trajectories and an initial state to output trajectories (see [Zeigler 2000] and [Mesarovic 1989]). A trajectory is simply a function from the system time base to a set of values. If we want to consider the values of a trajectory x over a range $[t_i,t_f]$ (or $[t_i,t_f)$ or $(t_i,t_f]$ or (t_i,t_f)), we can write $x[t_i,t_f]$ (or $x[t_i,t_f)$ or $x(t_i,t_f]$ or $x(t_i,t_f)$). The value of x at point t is given by x[t,t] and will often be written more conventionally as x(t). The output of an I/O functional system, for an initial state s_0, is given by $y[t_i,t_f] = F(s_0,x[t_i,t_f])$, where F is a function from a set of initial states and input trajectories to a set of output trajectories.

An I/O functional model of a system provides means for determining the output of a system that will result from applying a particular input to the system when it is in a particular state. In engineering and in many of the sciences, it is quite common to construct larger models from existing state space and I/O functional models (e.g., Newton's laws, conservation laws), thereby arriving at a state transition or coupled

component representation of the system being studied (e.g., see [Wymore 1993], [Karnopp 1975], [Zeigler 2000], [Cellier 1991]). 'Solving' such a system of equations amounts to reducing this system to an I/O functional representation.

For discrete event systems, we would like to formulate state transition and coupled component representations with a degree of precision that is taken for granted when dealing with models of discrete time and continuous systems. The DEVS atomic and coupled model specifications provide a structure for doing this, and their descriptions will set the stage for precisely formulating the simulation problem for discrete event systems.

2.2. Atomic Models

An atomic model is a system with a set of inputs, a set of outputs, a set of states, a set of transition functions, a time advance function, and an output function. Formally, an atomic model is defined by the structure

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

where

X is a set of inputs,

Y is a set of outputs,

S is a set of states,

$\delta_{ext}: Q \times X^b \rightarrow S$ is the external transition function,

where $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of total states,

e is the time elapsed since the last state transition, and

X^b is a bag of inputs whose elements are in X,

δ_{con}: $S \times X^b \rightarrow S$ is the confluent transition function,

δ_{int}: $S \rightarrow S$ is the internal transition function,

λ: $S \rightarrow Y$ is the output function, and

ta: $S \rightarrow R^+_{0,\infty}$ is the time advance function.

The external transition function describes how the system changes state in response to input. When input is applied to the system, it is said that an external event has occurred. The internal transition function describes the autonomous behavior of the system. When the system changes state autonomously, an internal event is said to have occurred. The confluent transition function determines the next state of the system when an internal event and external event coincide. The output function generates system outputs that coincide with internal events and whose value is determined by the current state of the system. The time advance function determines the amount of time that must elapse before the next internal event will occur, assuming that no input arrives in the interim. The range of the time advance function can be any suitable time base (see [Zeigler 2000]). The positive real numbers, including 0 and ∞, is an example of such a time base.

It is possible to define a DEVS model in such a way that it undergoes an infinite number of state transitions in a finite interval of time. Such a model is called illegitimate

because it does not specify a system (see [Zeigler 2000]). A model whose time advance function is defined to be zero for all states is the simplest example of an illegitimate model. Unless otherwise stated, any model that we considered is assumed to be legitimate.

The DEVS atomic model specification defines a state space representation of a system (see [Zeigler 2000] and [Mesarovic 1989]) in the following way. Let

$$S = (X, Y, Q, \Delta, \Lambda)$$

define the system of interest, where

X is an input set,

Y is an output set,

Q is a set of states,

$\Delta: Q \times X \rightarrow Q$ is the state transition function, and

$\Lambda: Q \rightarrow Y$ is the output function.

Let $x(t)$ be an event trajectory defined on the interval $[t_i,t_f]$ that takes the value of the non-event everywhere except at a finite number of points. The non-event will be denoted by the symbol Φ. Let $q = (s,e)$ be a state in Q, where s is a state of the DEVS model and e is an element of the time base. The state transition function for the DEVS model is defined recursively as follows (see [Zeigler 2000]):

2-3) $\Delta(q,x[t_i,t_f]) =$

case 1) $\Delta((\delta_{int}(s),0),x(t_i+ta(s)-e,t_f])$

if $e + t_f - t_i = ta(s)$ and $\forall t \in [t_i,t_i + ta(s) - e]$, $x(t) = \Phi$.

case 2) $\Delta((\delta_{ext}(s,e+t-t_i,x(t)),0),\Phi[t,t]\cdot x(t,t_f])$

if $\exists t \in [t_i, \min\{t_f,t_i + ta(s) - e\}]$ such that $x(t) \neq \Phi$

and $t \neq t_i + ta(s) - e$ and $\forall t' \in [t_i ,t), x(t') = \Phi$.

case 3) $\Delta((\delta_{con}(s,x(t_i+ta(s)-e)),0),\Phi[t_i+ta(s)-e,t_i+ta(s)-e]\cdot x(t_i+ta(s)-e,t_f])$

if $e + t_f - t_i = ta(s)$ and $\exists t \in [t_i, \min\{t_f,t_i + ta(s) - e\}]$ such

that $x(t) \neq \Phi$ and $t = t_i + ta(s) - e$ and $\forall t' \in [t_i ,t), x(t') = \Phi$.

case 4) $\Delta((s,e + t_f - t_i),x[t_i,t_f])$

if $e + t_f - t_i < ta(s)$ and $\forall t \in [t_i,t_i+e], x(t) = \Phi$.

The output function for the DEVS is given by:

2-4) $\Lambda(q) = \lambda(s)$ if $e = ta(s)$

Φ otherwise

The system described by an atomic DEVS model evolves through a series of internal (case 1), external (case 2), and confluent (case 3) events. The 'no event' (case 4) is included for completeness. An internal event occurs when an amount of time equal to the time advance has elapsed and no external events have occurred. An external event occurs when the input trajectory has a non-null value at some time prior to the the time advance expiring. A confluent event occurs when x(t) takes on a non-null value exactly when the time advance expires. The DEVS output function can take on non-null values only in conjunction with an internal or confluent event.

It should be noted that the definition causes input events to be consumed after being processed. This is accomplished by replacing the value x(t) by the non-event in the subsequent pass of the recursion (see cases 2 and 3).

The dynamic behavior of an atomic model can be generated by applying an abstract simulator to the model description. The abstract simulator is an algorithm that will generate a state and output trajectory for the atomic model given a model description and suitable input trajectory. For example, Algorithm 2.1 describes an abstract simulator for an atomic model [Chow 1994]. Let $x[t_0,t_f]$ be an event trajectory that we wish to apply to an atomic model. Assume that the atomic model begins in some initial state s_0. An output trajectory $y[t_0,t_f]$ is computed along the way. The output trajectory is assumed to initially take the value of Φ everywhere, and Algorithm 2.1 will assign values to particular points as the computation progresses.

Algorithm 2.1

```
1   tN := t0
2   tL := t0
3   s := s0
4   while (tN ≤ tf)
5       tN := min { tL+ta(s), t ∈ [tL,tf] such that x(t) ≠ Φ}
6       if (tN = tL+ta(s) and x(tN) ≠ Φ)
7               y(tN) := λ(s)
8               s := δcon(s,x(tN))
9               tL := tN
10              x(tN) := Φ
11      else if (tN = tL+ta(s) and x(tN) = Φ)
12              y(tN) := λ(s)
13              s := δint(s)
14              tL := tN
```

```
15      else if (tN < tL+ta(s) and x(tN) ≠ Φ)
16              s := δext(s,tN-tL,x(tN))
17              tL := tN
18              x(tN) := Φ
19      end
20 end
```

To illustrate Algorithm 2.1, let us consider the simulation of a simple counter. The counter accumulates inputs and, upon being incremented, provides the current value of the accumulator as an output. For the sake of illustration, we will assume only a single input element at any time (i.e., input bags are assumed to contain a single value). The counter can be described by a DEVS atomic model as follows:

$X = Y = Z,$

$S = Z \times \{1,\infty\}$ and we let (s,σ) denotes an element in S,

$\delta_{int}((s,\sigma)) = (s,\infty),$

$\delta_{ext}((s,\sigma),e,x) = (s+x,1),$

$\delta_{con}((s,\sigma),x) = (s+x,1),$

$ta((s,\sigma)) = \sigma$, and

$\lambda((s,\sigma)) = s.$

Let the initial state of the counter be $(0,\infty)$ and consider the input trajectory x[0,2] where $x(t) = \Phi$ when $t \neq 1$ and $t \neq 2$, $x(1) = 1$, and $x(2) = 1$. At line 5 we compute $tN = \min\{\infty,1\} = 1$ and have that $x(tN) = 1$. This causes line 15 to evaluate to true and so we compute the next state $s = \delta_{ext}((0,\infty),1,1) = (1,1)$ and set $tL = 1$ and $x(1) = \Phi$.

On the next iteration of Algorithm 2.1, we compute, at line 5, tN = min {2, 2} = 2 and x(tN) = 1. This causes line 6 to evaluate to true and so we compute the next state s = $\delta_{conf}((1,1),1) = (2,1)$. We set y(2) = 1, x(2) = Φ, and tL = 2.

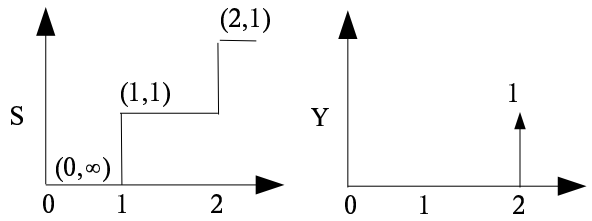

Figure 2.1. State trajectory (S) and output trajectory (Y) for the counter model.

At this point we have computed the state trajectory and output trajectory shown in Figure 2.1. It can be readily verified that these trajectories are correct.

The abstract simulator has several properties of interest. First and foremost is that the algorithm processes only and all correct internal, external, and output events. That is, it processes exactly those events that are required by the modeling formalism given the initial state and input trajectory being applied to the model, and in doing so it produces exactly the required output events. This property will be called the **correct event** property. The correct event property implies that all events are retained, all events are properly interpreted, and no spurious events are introduced.

More formally, the correct event property can be stated as follows: let $x[t_0,t_f]$ be an input trajectory that we apply to a model in some initial states $s(t_0)$. Let $s[t_0,t_f]$ and $y[t_0,t_f]$ be the resulting state and output trajectories that are produced by functions (2-3) and (2-4) respectively. Let $s'[t_0,t_f]$ and $y'[t_0,t_f]$ be the state and output trajectories produced by a simulation algorithm. The correct event property holds for the algorithm if $s[t_0,t_f]$ =

$s'[t_0,t_f]$ and $y[t_0,t_f] = y'[t_0,t_f]$. When the correct event property holds for an execution of an algorithm, we will say that the algorithm has computed all and only correct events.

While the correct event property is implicit in the work of [Ghosh 1996] and is explicitly stated by [Bagrodia 1991], it is not treated as a fundamental property of a simulation algorithm by most authors. In fact, [Ghosh 1996] does not show that the correct event property holds, but rather constructs a proof that events are processed in time stamp order. That the correct events are executed is assumed (not shown) to be true.

Similarly, [Bagrodia 1991] shows that it is possible for a particular execution of an algorithm to adhere to the correct event property. However, the non-determinism of the base algorithm (i.e., the space-time algorithm) from which the particular algorithm is derived seems to indicate that different restrictions of the base algorithm can produce different outcomes given identical systems, input trajectories, and identical states. How can each of these outcomes be correct? Since it is possible for some execution of the base algorithm to produce a correct result (i.e., the correct event property holds), there must exist some restriction of the base algorithm that produces a correct result. The question of which restriction is needed for a particular system is left unanswered.

It can be readily demonstrated that executing events in the correct order is not sufficient to guarantee that the correct event property holds. Consider, for example, the following 'simulation algorithm'. Suppose that we are given a set of events e_1, e_2, ... e_n with time stamps $t_1 < t_2 < ... < t_n$ respectively. To 'simulate' this system, execute event e_{n-1} followed by e_n. Since $t_{n-1} < t_n$, our 'simulation' algorithm executes events in the correct order. However, the algorithm discards events e_1, ... , e_{n-2} and so the correct event

property is violated. Another obstacle that arises when trying to adapt existing algorithms for simulating general systems is a sudden shift in focus from functions to event sequences, and this is addressed more fully later on in this text.

Demonstrating that the correct event property holds is the goal of the correctness proofs that will be presented later. In constructing these proofs, it is sometimes useful to demonstrate several other properties that, intuitively, would seem to precede the correct event property. Many of these intermediate properties are meaningful only in the context of a distributed computing system, and these properties will be introduced as needed.

One property which has received considerable attention can be introduced immediately. This is the time stamp order property. This property holds if events are processed in time stamp order. That is, we compute events in order, beginning at some initial time and proceeding along the time axis in an orderly way until the appropriate end time is reached.

Many researchers who have addressed the correctness of parallel simulation algorithms (e.g., see [Ghosh 1996] and [Frey 2002]) have presented proofs of time stamp order event processing, freedom from deadlock, and other (important!) properties of parallel algorithms. The correct event property (i.e., that the algorithm correctly simulates the model) has not been addressed. This fact has been recognized (e.g., see [Ghosh 1996], [Bagrodia 1991], and [Page 1994]), and it has been attributed to an inability to identify the model being simulated or to state precisely what correct means when a model has been identified. However, a handful of authors have considered

simulation protocols in which the correct event property is known to be violated (see, e.g., [Reynolds 1988][Rao 1998][Thondugulam 1999]).

The use of a modeling formalism solves the problem of model identification so long as we restrict ourselves to systems that can be represented in the formalism. We bypass the latter issue (which is really an issue of model validation) by addressing the correctness of an algorithm with respect to a modeling formalism and its abstract simulator in particular. This amounts to proving that an algorithm is correct for a class of systems (i.e., the systems that can be expressed in the formalism). Given a model expressed in terms of a particular system formalism, we can expect a simulator that is correct with respect to that formalism to generate the proper model behavior.

A correctness proof for Algorithm 2.1 will serve to illustrate a general method of proof that will be employed throughout this text (see [Tel 2000]). The proof is inductive. The base case is where the initial and final simulation times are identical and so there are no events to process. The induction hypothesis is that the computation has proceeded correctly (i.e., exhibited the correct event property) until some intermediate time has been reached. The inductive step consists of showing that the next output is correctly computed (i.e., the correct event property holds after the next event is processed). The inductive step is done by case analysis. There are four cases; an internal event, an external event, a confluent event, and no event.

Theorem 2.1. Consider a DEVS model with initial state s_0 to which an event trajectory $x[t_0,t_f]$ is applied. The correct event property holds for Algorithm 2.1 when it is applied to that model.

Proof. Suppose that $t_0 = t_f$. Then line 3 evaluates to false and no computation is performed. Since there were no events to compute, this is correct. Also note that the last event time tL equals the next event time tN. So the elapsed time e = tN – tL = 0.

Now, suppose that all and only correct events have been processed up to some time t in the interval $[t_0,t_f]$ and that e = tN – tL = 0 at line 4. Consequently, we can assume that t_i in function (2-3) corresponds to tL in Algorithm 2.1. Let s be the system state at that time. The next correct event can be an internal event, an external event, or a confluent event. The no event case is not of interest since it would require that, at line 4, tN take a value other than one of the values in the set over which we are finding a minimum.

Case 1. Suppose that the next correct event is an internal event. By hypothesis, e = tN – tL = 0. From the definition of the DEVS transition function given by (2-3), it is readily determined that the time of next event $t = t_i + ta(s) < \min \{ t' \in [t_i,t_f]$ such that $x(t') \neq \Phi \}$. By the induction hypothesis, at line 5 we have tN = tL+ta(s) and $x(tN) = \Phi$. Consequently, line 10 evaluates to true and we assign, first, $y(tN) = \lambda(s)$ as is required by the definition of the output function. Next, the subsequent state is computed using the internal transition function. Lastly, the time of last event is updated so that e = tN – tL = 0. No other commands are executed prior to beginning the next computational cycle at line 4.

Case 2. Suppose that the next correct event is an external event. From the definition of the DEVS transition function, it is readily determined that the time of next

event t = min { t' ∈ $[t_i,t_f]$ such that x(t') ≠ Φ } < t_i + ta(s). By the induction hypothesis, tN = min { t' ∈$[tL,t_f]$ such that x(t') ≠ Φ } < tL+ta(s) and x(tN) ≠ Φ. In this case, line 14 evaluates to true. We then compute the next state using the external transition function with the elapsed time e = tN – tL > 0. The elapsed time is updated by setting tN = tL and so e = tN – tL = 0 at line16. Lastly, the input event at time tN is consumed by setting x(tN) to the null-event. No other commands are executed prior to beginning the next computational cycles at line 4.

Case 3. Suppose that the next correct event is a confluent event. From the definition of the DEVS transition function, it can be seen that the time of next event t = min { t' ∈$[t_i,t_f]$ such that x(t') ≠ Φ } = t_i + ta(s). By, the induction hypothesis, tN = min { t' ∈$[tL,t_f]$ such that x(t') ≠ Φ } = tL + ta(s) and x(tN) ≠ Φ. So line 6 evaluates to true. First, we set y(tN) = λ(s) as is required by the definition of the output function. Next, the subsequent state is computed using the confluent transition function. The time of last event is updated so that e = tN – tL = 0. Lastly, the input event at time tN is consumed by setting x(tN) to the null-event. No other commands are executed prior to beginning the next computational cycle at line 4. ■

2.3. Coupled Models

A coupled model consists of a set of component models and their interconnections (couplings). The components can be atomic models or other coupled models. The

coupled model's behavior is defined by the behavior and interconnectedness of its components. Component couplings are divided into three types: external input coupling, external output coupling, and internal coupling. External input coupling occurs between the coupled model itself and the inputs to one or more of its components. External output coupling occurs between component outputs and outputs of the coupled model. Internal coupling occurs between component outputs and components inputs. Figure 2.2 depicts these possibilities.

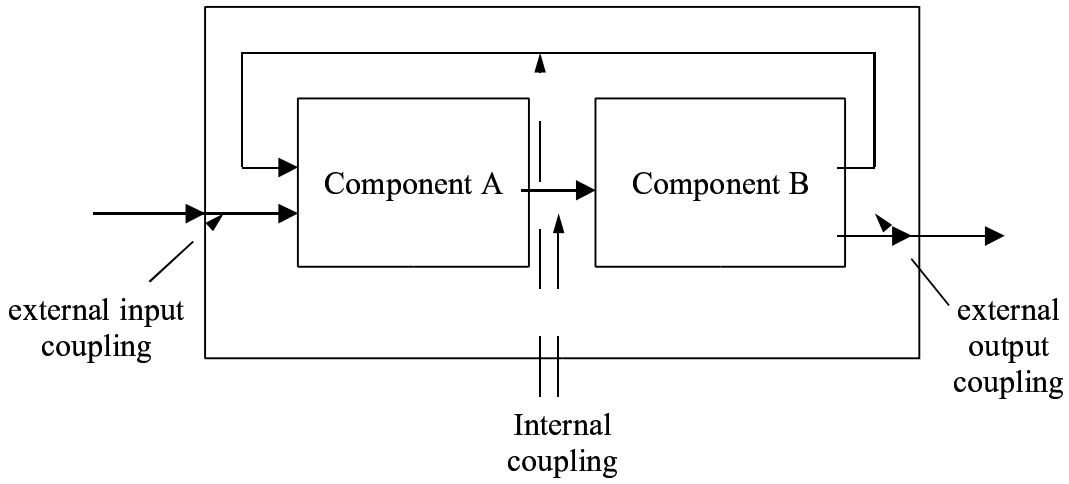

Figure 2.2. Valid couplings between components.

Formally, a coupled model is described by a structure

$$DN = (X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}),$$

where

X is a set of inputs,

Y is a set of outputs,

D is a set of component references,

For each $i \in D$, M_i is an atomic structure,

For each $i \in D \cup \{self\}$, I_i are the components influenced by i, and

$Z_{i,j}$ is the i-to-j output translation function where

$Z_{self,,j}$: $X_{self} \rightarrow X_j$ is the external input coupling function,

$Z_{i,self}$: $Y_i \rightarrow Y_{self}$ is the external output coupling function, and

$Z_{i,j}$: $Y_i \rightarrow X_j$ is the internal coupling function.

The coupled DEVS structure is subject to the constraint that, for each i in D,

1) I_i is a subset of $D \cup \{self\}$ and i is not in I_i

2) The only functions in $Z_{i,j}$ correspond to external input, external output, and internal couplings between components.

Closure under coupling is a property of coupled models that allows any coupled model to be reduced to a behaviorally equivalent atomic model. An important aspect of the closure under coupling property is that the coupled model and its atomic counterpart are indistinguishable if viewed as I/O functional systems (see [Zeigler 2000]).

In this text, we will be concerned primarily with simulating a single coupled model whose components (either coupled or atomic models) are simulated in parallel. More specifically, we will focus on distributing the components of the 'top-level' model. With this goal in mind, let us try to construct an algorithm for simulating coupled models that is readily seen to be correct, and yet is simple and abstract enough to provide a starting point for constructing new algorithms.

Assume that every component model is an atomic model or a reduction of a coupled model to a behaviorally equivalent atomic representation. For any particular

component, if we are given an input trajectory to apply to the component, we can use Algorithm 2.1 to compute the resulting output trajectory. When a coupled model is simulated, its behavior is determined by applying an input trajectory to each component model. Unlike Algorithm 2.1, however, the input trajectories are not known *a priori* and must be constructed as the simulation progresses. Let us suppose that the intended result of simulating a coupled model is to apply an input trajectory $x_i[t_0,t_f]$ to the i^{th} component of the coupled model. How must the x_i be defined?

Let $y_i[t_0,t_f]$ be the output trajectory that results from applying $x_i[t_0,t_f]$ to the i^{th} component. Letting $\sum$ be the bag union function over a set of bags, we must have that

$$x_i(t) = \sum_{j \in D, j \neq i} Z_{j,i}(y_j(t)). \qquad 2\text{-}5)$$

The primary problem in constructing a simulator for coupled models is to ensure that the simultaneous construction of input and output trajectories is done correctly. By correct, we mean that at the end of a simulation run (i.e., when the simulation clock has reached t_f) the output trajectories generated by each component model are exactly those that would result if the the input trajectory given by (2-5) was applied to the component via Algorithm 2.1.

Algorithm 2.2 describes a procedure for simulating a coupled model with atomic components [Chow 1994]. As in Algorithm 2.1, we let tL denote the time of last event and tN the time of next event. The subscripted x and y functions are the input and output functions, respectively, of the component denoted by the subscript.

Algorithm 2.2

1 tL := t_0
2 for each i ∈ D
3 $x_i[t_0,t_f] := \Phi[t_0,t_f]$
4 end
5 while (tN ≤ t_f)
6 for each i ∈ D
7 $y_i[t_0,t_f]$:= output trajectory produced by Algorithm 2.1 given input $x_i[t_0,t_f]$
8 end
9 tN := min { t : t > tL and, for some i ∈ D, $y_i(t) \neq \Phi$}
10 for each i ∈ D
11 $x_i[t_0,t_f] := x_i[t_0,tN) \cdot \sum_{j \in D,\, j \neq i} Z_{j,i}(y_j(tN)) \cdot \Phi(tN,t_f]$
12 end
13 tL := tN
14 end

To illustrate Algorithm 2.2, consider a coupled model whose components are the counter from the previous example and a generator. The coupled model has a set of component names D = {c,g} where c denotes the counter and g denotes the generator. The components are connected in a simple feedback arrangement. The coupling functions are defined such that $Z_{c,g}(q) = q$ and $Z_{g,c}(q) = q$.

The generator is defined as follows:

$X = Y = Z$,

$S = Z \times \{1,\infty\}$ and we let (s,σ) denote an element of S,

$\delta_{int}((s,\sigma)) = (s,1)$,

$\delta_{ext}((s,\sigma),e,x) = (x,\gamma)$,

where $\gamma = 1$ if $x = 0$ and $\gamma = \infty$ otherwise

$\delta_{con}((s,\sigma),x) = \delta_{ext}((s,\sigma),0,x)$,

$ta((s,\sigma)) = \sigma$, and

$\lambda((s,\sigma)) = 1$.

Let us simulate this coupled model over the interval [0,2]. When we invoke Algorithm 2.1, let the initial state of the generator be $s_g = (0,1)$ and the initial state of the counter be $s_c = (0,\infty)$. At line 3, we construct the input trajectories $x_g[0,2] = x_c[0,2] = \Phi[0,2]$. At line 7 we use these input trajectories and Algorithm 2.1 to construct the output trajectories $y_g[0,2]$ and $y_c[0,2]$. The trajectory $y_g[0,2]$ has values $y_g(0) = \Phi$ and $y_g(1) = y_g(2) = 1$. The trajectory $y_c[0,2] = \Phi[0,2]$.

Using these output trajectories and the fact that $tL = 0$, at line 9 we compute $tN = 1$. So at line 11 we construct two new input trajectories for the counter and the generator. The generator input trajectory does not change, so $x_g[0,2] = \Phi[0,2]$. The counter input trajectory is redefined such that $x_c(0) = x_c(2) = \Phi$ and $x_c(1) = 1$. At line 13, we set $tL = 1$ and first iteration of the algorithm is completed.

On the second pass of the algorithm, we first construct two new output trajectories at line 7. The generator output trajectory is constructed using the input trajectory $x_g[0,2] = \Phi[0,2]$ which was computed in the previous iteration. This results in a generator output trajectory that takes on values $y_g(0) = \Phi$ and $y_g(1) = y_g(2) = 1$. The counter output trajectory is computed using the input trajectory whose values are $x_c(0) = x_c(2) = \Phi$ and $x_c(1) = 1$. This results in an output trajectory with values $y_c(0) = y_c(1) = \Phi$ and $y_c(2) = 1$.

At line 9, we compute tN = 2 using the output trajectories computed at line 7 and the fact that tL = 1. At line 11, we compute a new pair of input trajectories. The generator input trajectory takes on values $x_g(0) = x_g(1) = \Phi$ and $x_g(2) = 1$. The counter input trajectory takes on values $x_c(0) = \Phi$ and $x_c(1) = x_c(2) = 1$. At line 13, we set tL = 2 and the algorithm terminates.

The outcome of this simulation agrees exactly with the previous example with respect to the computed behavior of the counter. In this case, however, the input trajectory for the counter was constructed simultaneously with the input trajectory for the generator. By studying the example, it can be seen that function (2-5) is computed incrementally using Algorithm 2.2 at each step. That is, having computed the values of the input trajectories up to time tL, we use Algorithm 2.1 and the coupling functions for the coupled model to find the value at time tN for each input trajectory. We then set tL to tN and repeat. That the values of tN and $y_i(tN)$ are correct is ensured by the fact that x_i is correct up to tL and that Algorithm 2.1 is correct.

Algorithms 2.1 and 2.2 serve to highlight two criteria that will be used to asses the correctness of any DEVS simulators that we construct. First, we assume some input trajectory that is correct to time t for a single component and show that the correct event property holds. Using the fact that a correct input trajectory leads to a correct output trajectory, we then show that the algorithm computes function (2-5) at the next event time which is greater than t. That is, we show that the Algorithm 2.2 correctly computes a set of I/O trajectories incrementally. The first aspect (i.e., correct simulation of a single

component) is illustrated by Theorem 2.1. Theorem 2.2 will provide an illustration of the second aspect.

Theorem 2.2. Consider a coupled DEVS model whose components have some known initial state. Algorithm 2.2 computes function (2-4) for each component model.

Proof. In Algorithm 2.2, we use Algorithm 2.1 to simulate any single component. Theorem 2.1 asserts that the correct event property holds for Algorithm 2.1. Assume that at line 5 in Algorithm 2.2 the x_i have been computed using correct y_j up to time tL. So at line 7 we compute output trajectories that are correct with respect to the input trajectories. Since we assume that the x_i are correct up to tL, and the y_i are correct with respect to the x_i, the value of tN is the correct next event time, so long as the the correct x_i in the interval (tL,tN) are null-valued.

Suppose for some x_i and for some $t \in (tL,tN)$ that $x_i(t) \neq \Phi$. This would require that some $y_j(t) \neq \Phi$. However, if that were the case, then at line 9 we would find that t = tN. This contradicts the assumption that t < tN. Consequently, tN is the correct next event time. At line 11, we use (2-5) to compute the trajectory $x_i[t_0,tN]$. After setting tL = tN, we see that $x[t_0,tL]$ has been computed using (2-5). Thus, Algorithm 2.2 is a correct simulator. ■

3. ALGORITHMS FOR PARALLEL DISCRETE EVENT SIMULATION

Existing parallel algorithms for simulating discrete event systems have been grounded in models of distributed computing systems rather than in systems theory. Moreover, the logical process model of a discrete event system, which forms the basis for most parallel simulation algorithms, is closely related to the event sequence model used to describe distributed computing systems. As such, it is helpful to have on hand some necessary background material in distributed computing before delving into a discussion on parallel discrete event simulation. This chapter presents, in a very basic form, parallel computing techniques which are integral components of the parallel simulation algorithms that will be developed later on in this text. A more thorough treatment of this material can be found in [Garg 1996], [Tel 2000], [Fujimoto 2000], [Mattern 1993], and [Mattern 1999].

3.1. A Model for Distributed Computing Systems

The model presented here is taken from [Garg 1996]. The model assumes a set of uniquely identified processors that communicate via messages. The execution of a program on a processor is modeled by a series of events. Events are categorized as internal events, send events, and receive events. An internal event does not involve communication with another processor. A send event causes a message to be transmitted by the processor. A receive event models the receipt of a message by the processor.

For each processor, there is a finite sequence of events that describes the execution of a program on that processor. A distributed execution consists of several such event sequences that are related by send-receive event pairs. A space-time diagram is used to visually depict the execution of a distributed program, and consists of a horizontal line for each processor that denotes (real) time at that processor. Along the horizontal lines are dots that correspond to events. A dot with an arrow entering it depicts a receive event, while a dot with an arrow leaving it depicts a send event. A dot without an arrow is an internal event. This is illustrated in Figure 3.1.

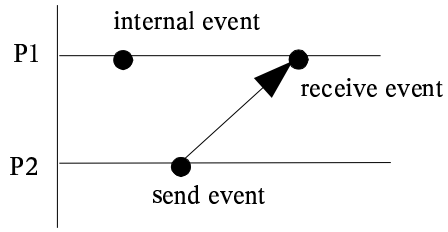

Figure 3.1. A space-time diagram for a two processor distributed system.

Informally, an event e is said to causally precede an event f if a path moving forward in time can be traced from e to f in a space-time diagram of the program execution. This is made precise in Definition 3.1

Definition 3.1. An event e is said to immediately precede an event f if

i) e and f are executed on the same processor and e is the event that appears immediately before f in the execution trace of the processor, or

ii) e is a send event and f is the corresponding receive event.

The causal precedence relation is the transitive closure of the immediately precedes relation.

If an event e causally precedes and event f, this is denoted by $e \rightarrow f$. When it is not the case that $e \rightarrow f$ and it is not the case that $f \rightarrow e$, then e and f are said to be causally independent. This is denoted by $e \parallel f$.

3.2. Logical Clocks

Logical clocks are a tool for detecting causal precedence relations in a distributed computing environment (see [Lamport 1978] for the original presentation, but it has since appeared in innumerable other works). Generally speaking, it is not necessary to have information about every causal chain that appears in the execution of a distributed program. This is especially true for distributed simulation, where we will be concerned with answering the question *could event e have caused event f?* This question is relevant to optimistic simulation algorithms in which causal incorrect calculations can occur, but they are corrected using a checkpoint and restore mechanism whenever the answer to *could event e have caused event f* is affirmative. In fact, we can often tolerate false positives (but not incorrect negatives) without compromising the correctness of a simulation algorithm as this will only result in an unnecessary, but not incorrect, checkpoint restoration.

It should be noted that the question *could event e have caused event f* is not the same as asking *did event e happen before event f.* For example, suppose that the two events occur in distinct systems that are not coupled in any way (i.e., do not

communicate). Then the answer to the first question is no while the answer to the second question could very well be yes.

A logical clock is a function from a set of events to a totally ordered set of time values. The function is constructed in such a way that the ordering of the times assigned to events reflects some aspect of the causal ordering of the events. There are three general types of clocks. These are minimally consistent clocks, weakly consistent clocks, and strongly consistent clocks.

A minimally consistent clock is described by Definition 3.2 (see, e.g., [Rönngren 1999]).

Definition 3.2. Let e_1 and e_2 be two events and let the function C be 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 $\neg(e_1 \rightarrow e_2)$. In the context of a discrete event simulation, a minimally consistent clock can be readily obtained using the times associated with events. A minimally consistent clock can provide information about causal relationships between events with different time stamps, but it can not provide information about events that have the same time stamp (i.e., it can happen that e_1 causes e_2 but the clock readings for these events are the same).

A weakly consistent clock assigns different times to all causally related events. A weakly consistent clock is described by Definition 3.3 (see [Garg 1996], [Mattern 1999]).

Definition 3.3. Let e_1 and e_2 be two events and let the function C be a weakly consistent clock. If $e_1 \rightarrow e_2$ then $C(e_1) < C(e_2)$.

From the definition, it can be seen that

i) If $C(e_1) \geq C(e_2)$ then $\neg(e_1 \rightarrow e_2)$, and

ii) If $C(e_1) = C(e_2)$ then $e_1 \parallel e_2$.

Statement (i) is simply the contrapositive of Definition 3.3. Thus if e_1 has a time stamp greater than or equal the time stamp assigned to e_2, then e_1 can not causally proceed e_2. Statement (ii) is obtained by noting that $C(e_1) = C(e_2)$ implies $C(e_1) \geq C(e_2)$ and $C(e_2) \geq C(e_1)$ and then applying (i). Thus if e_1 and e_2 have equal time stamps, then e_1 and e_2 are causally independent. However, notice that if e_1 has a time stamp less than that of e_2, it may still be that they are causally independent. The next definition mitigates against this posibility.

A strongly consistent clock meets the strong clock condition described in Definition 3.4 (see [Garg 1996], [Mattern 1999]).

Definition 3.4. Let e_1 and e_2 be two events and let the function C be a strongly consistent clock. In this case, $e_1 \rightarrow e_2$ if, and only if, $C(e_1) < C(e_2)$.

The use of strongly consistent clocks in distributed simulation is considered in [Rönngren 1999]. For our purposes, as mentioned earlier, we can tolerate false positives. Therefore a weakly consistent clock is sufficent. Furthermore, the additional effort in

terms of communication overhead and algorithm complexity required to apply the strong clock condition in simulation makes it unattractive for practical application.

3.3. Consistent Cuts

Let E denote the set of all events that occur during the run of a distributed program. If we order E using the causal precedence relation, then a consistent cut partitions E into a set of past and future events with respect to the causal precedence relation. An inconsistent cut of E partitions E in such a way that the future partition contains an event that causally precedes an event in the past partition. Figure 3.2a is an example of a consistent cut, while Figure 3.2b demonstrates an inconsistent cut.

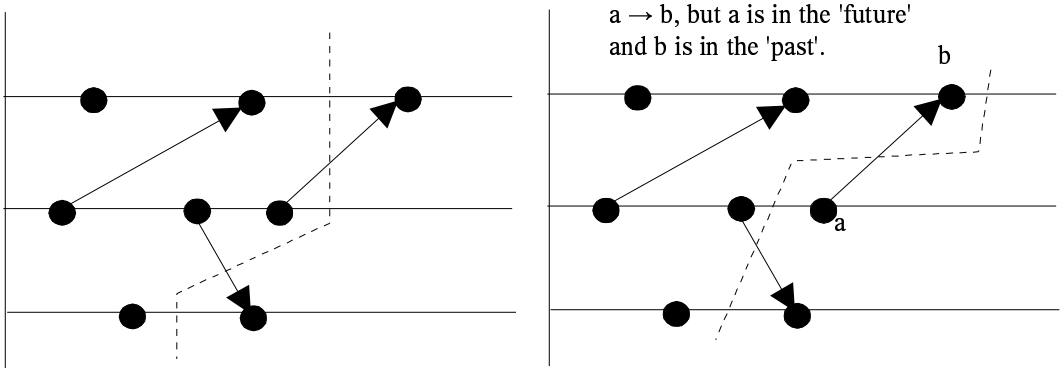

Figure 3.2. A consistent cut (3.2a on left) and an inconsistent cut (3.2b on right).

Suppose we apply a weakly consistent clock to the set E and then find a consistent cut such that no initial event is in the future partition and there are no send events in the past partition whose corresponding receive event rests in the future partition. Then the smallest time assigned to an event in the future partition is strictly larger than the smallest time assigned to an event in the past partition. In the context of discrete event simulation,

the smallest time assigned to an event in the future partition is called the global virtual time. The global virtual time places a lower bound on the time at which any future event can occur.

In order to compute the global virtual time, we need to ensure that no messages are in transit when the cut is being computed (i.e., there are no send events in that past with corresponding receive events in the future). One solution is to devise an algorithm that pushes the cut line to the right in such a way that no message crosses the cut line. For example, Figure 3.3 shows the consistent cut in Figure 3.2a pushed left to account for messages that cross the cut line.

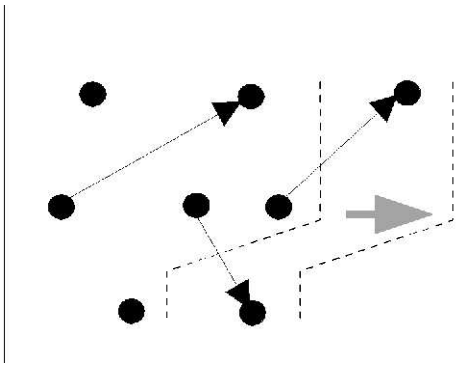

Figure 3.3. A consistent cut that accounts for all messages crossing the cut line.

An algorithm for finding consistent cuts that accounts for messages in transit is listed as Algorithm 3.1 (see, [Garg 1996]). Each process is uniquely identified and includes its process identification with every message that it sends. Processes can send a special message, called a marker, that is used to account for messages in transit when the algorithm is executed. Messages in transit when the cut algorithm is started are, in effect, flushed out of the communication system by the marker. The receiving process will

collect these flushed messages in a queue. Each process also has a color, which is initially white.

Some process is assumed to spontaneously start the cut algorithm. When a process executes the cut algorithm, it turns red and sends a marker to all of the processes that are adjacent to it. The receiver executes the cut algorithm on receipt of the marker, thereby turning red and sending a marker to its neighbors. The cut has been found when all of the processes turn red. Any particular process can determine that its part in the cut computation has ended if all of the processes that can send it messages have provided markers.

In general, we are not interested directly in the set of all events that are captured in the past and future partitions of the cut. Rather, we are interested in some property of the distributed system that can be inferred from the partitioning. For example, to determine the global virtual time in a parallel discrete event simulations it is necessary to find the smallest time stamp assigned to events in the future partition of the cut.

As such, when a process turns red it will record some relevant piece of state information that will be of use in determining the property of interest. If we are calculating global virtual time, this would be the time of the next event at each process. In Algorithm 3.1, the point at which the relevant portion of the process state should be recorded is denoted by "record current value of pertinent state variables".

When the cut algorithm terminates, every process will have recorded its state (i.e., a record of all events processed in the past partition). The unprocessed events that were in transit when the cut was started will be stored in queues at the receiving processes.

When implementing distributed simulation algorithms, this information will be used to compute the global virtual time (see 3.5) by taking the minimum of the time stamps of the stored messages and the local time of next event.

Algorithm 3.1

Let P_{in} be the set of processors that can send messages to this process.
Let P_{out} be the set of processors that can receive messages from this process.
Let color $\in$ { red, white} with the initial value white.
Let closed[k] with k $\in P_{in}$ be boolean values where, initially, closed[k] = false for all k.
Let chan[k] with k $\in P_{in}$ be lists of messages where, initially, chan[k] is empty for all k.

```
1.  function turn_red
2.        color := red
3.        record current value of pertinent state variables
4.        send marker to all processes in Pout
5.  end turn_red
6.
7.  when receive marker from process k
8.        if (color = white) turn_red
9.        closed[k] := true
10.end
11.
12.when receive message that is not a marker from process k
13.       if (closed[k] = false) add message to chan[k]
14.end
```

3.4. Computing Minimum Values

Suppose we have a finite set of values and we want to find the minimum value in that set. Moreover, assume that each process in the distributed system contains some subset of values in the set such that the union of value sets held by all processes gives the value set of interest. A minimum value for such a distributed set can be found using two

passes of any wave algorithm (see [Tel 2000]). This is illustrated with two examples. Algorithm 3.2 provides an example of a minimum finding algorithm that could be implemented on a network with a ring topology. Algorithm 3.3 computes a global minimum when the processors are organized as a binary tree.

Algorithm 3.2 operates as follows. One process starts the minimum computation. That process computes the minimum of its locally held values and then sends the result to the left. The initiating process then waits for a message from the right that will contain the global minimum. At this point, the minimum is known and needs to be distributed to all of the processes. To do this, the initiating process sends the final result left.

All other processes wait for a message from the right. When a message is received, the value contained in that message is added to the set of locally held values, and then the process finds the minimum in that set. The result is sent to the left. The process then waits to receive a message containing the global minimum from the right. When this message is received, it is relayed to the left. The algorithm finishes when each process has received the global minimum from the right.

Algorithm 3.2

Let p_{left} and p_{right} be the processors to the left and right, respectively, of this processor.
Let V be the set of values held by this processor.
Let start $\in$ { true, false } be true if this processor starts the computation and false otherwise.

1. if (start = true)
2. min_value := min V
3. send min_value to p_{left}
4. wait for result from p_{right}
5. else

6. wait for min_value from p_{right}
7. min_value = min V ∪ {min_value}
8. send min_value to p_{left}
9. wait for result from p_{right}
10. end

Algorithm 3.2 computes a minimum serially, with each processor performing its part of the calculation in turn. For connection topologies in which a tree can be constructed efficiently (e.g., a cluster of workstations with point to point connections between nodes) it is possible to compute a global minimum in parallel. Given a set of processors labelled 1, 2, ... , N a tree can be constructed as shown in figure 3.4.

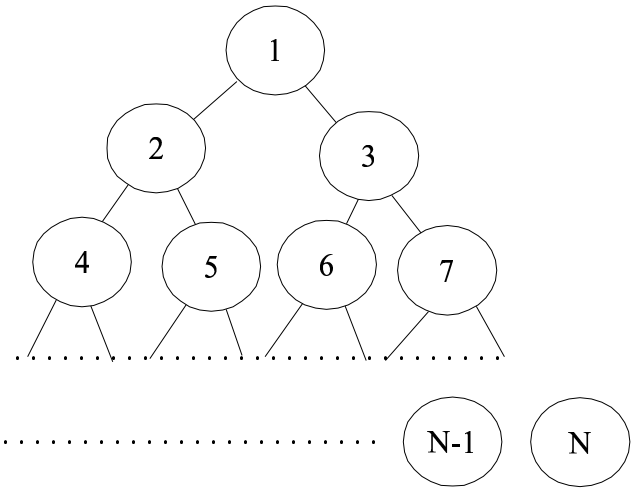

Figure 3.4. A binary tree used to compute global minimums on a parallel computer.

Algorithm 3.3 shows the details of computing a minimum on this tree. The leaf nodes start the computation by sending their value to their parents. On receiving a value from each child, the parent finds the minimum of its own value and the values supplied by its children. The result is then set to the parent of this node and this is repeated until

we reach the root. At this point, the root has the global minimum which it broadcasts to every node in the system. The broadcast can be implemented directly on the tree or using any other efficient broadcast mechanism.

Algorithm 3.3

```
1.  if (this node is the root)
2.          wait for values from children
3.          min_value := min of children values and this node's value
4.          broadcast min_value to all nodes
5.  else if (this node is a leaf)
6.          send this node's value to this node's parent
7.          wait for global minimum to arrive from root
8.  else
9.          wait for values from children
10.         min_value := min of children values and this node's value
11.         send min_value to this node's parent
12.         wait for global minimum to arrive from root
13.end
```

3.5. Global Virtual Time

The notion of global virtual time is essential to optimistic discrete event simulation algorithms. The global virtual time of a distributed simulation is the smallest of the time stamps associated with unprocessed events (i.e., it is the global time of next event). Given a weakly consistent clock, Algorithm 3.1 (or any other algorithm for finding consistent cuts), and a global minimum algorithm such as Algorithm 3.2 or Algorithm 3.3, we can build an algorithm for computing global virtual time. The problem of finding global virtual time has been addressed by several authors (see, for example, [Tomlinson 1993], Mattern [1993], Fujimoto [1997]), and a wide variety of

schemes are described in the literature. The algorithm presented here is simplistic, but it serves to illustrate the essential concepts.

Initially, some process decides to compute the global virtual time. This process initiates a cut by executing Algorithm 3.1. The relevant state information is the time of next event at the process. The messages that are collected in the chan[k] lists are simulation events. When the cut has been found, each process computes its local virtual time by taking the minimum of the time stamps in each chan[k] list and its own time of next event.

Having found a consistent cut, a minimum finding algorithm (e.g., Algorithm 3.2) is used to find the smallest time held by any process in the system. This smallest time is the global virtual time. These steps are listed as Algorithm 3.4.

Algorithm 3.4

LVT is the local time of next event (Local Virtual Time)
GVT is the global time of next event (Global Virtual Time)

1. find a consistent cut. Save the local time of next event and time stamps of events stored in the chan[k] lists to the set VT.
2. LVT := min VT
3. find GVT by computing a global minimum value using LVT as the sole element in the local value set

This chapter has introduced the elements of parallel computing that are needed to implement the optimistic simulation algorithms that will presented later in this dissertation. What remains is to link the concepts developed in this chapter, and especially the concept of causal precedence, to the discrete event systems framework that

our simulation algorithms are meant to support. Once this final step is complete, we will be able to put the material developed here to work simulating discrete event systems.

4. CAUSALITY AND CAUSAL PRECEDENCE

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., see [Garg 1996] and [Tel 2000]). 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 (see, for example, [Garg 1996]). This formalization is useful for proving that parallel 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 ([Reed 1987],[Fujimoto 2000]). 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.

One difficulty with this approach is that, 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 (see, e.g., [Zeigler 1999]). Examples of such models are partial differential equations and systems of ordinary differential equations approximated by quantized state systems [Zeigler 2000]. While these models might be addressed on a case-by-case basis, e.g., the discrete time algorithms presented in [Bagrodia 1991] and

[Fujimoto 1998], this presents considerable difficulty when we wish to demonstrate the utility of an algorithm for simulating a class of systems. Another, more generally applicable, approach is to map system-theoretic formalisms into a logical process world-view. If this mapping can be developed for a class of systems (in our case, the class of systems with DEVS representations), then it allows us to construct simulation environments whose correct operation can be ensured for the class of systems covered by the mapping.

This chapter developes the foundations of a conceptual framework for mapping formal and semi-formal modeling constructs into a logical process world-view. The conceptual framework itself is presented in the chapter that follows. The significance of this foundational work is that it establishes a notion of time and causality that will be essential for constructing demonstrably correct parallel simulation algorithms. This chapter first presents background material related to causality in the abstract time systems theory (see [Mesarovic 1989] and [Zeigler 2000]). The problems of time and causality in distributed simulation systems are then revisited in the context of simulating causal time systems.

4.1. Systems, Causality, and Causal Precedence

An I/O relational system is a system for which the input quantities and output quantities can be distinguished ([Mesarovic 1989],[Zeigler 2000],[Klir 1969]). A relational system S is described as a subset of 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.

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.

It can be convenient to consider abstract time systems over finite intervals of time. An interval is described using the notation presented in Chapter 2. A right half interval for an abstract time function x is described using the notation x|t) (or x|x]). This notation indicates that we are considering values of time less than t (or less than or equal to t). A restriction of a function is the function obtained by restricting the domain of the original function to a smaller, finite interval.

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, two notions of causality exist [Mesarovic 1989]. 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.

Definition 4.1 gives a formal definition of causality for abstract time systems [Mesarovic 1989].

Definition 4.1 An abstract time system $S \subset X \times Y$ is causal if

i) $x|t] = x'|t] \Rightarrow y|t] = y'|t]$.

The system is strongly causal if condition (i) is replaced by the stronger condition

ii) $x|t) = x'|t) \Rightarrow y|t] = y'|t]$.

The following properties are direct consequences of Definition 4.1.

Property 4.1 If S is a causal system, then S with inputs restricted to $[t_0,t_f]$ and outputs restricted to $[t_0,t_f]$ is a function.

Property 4.2 If S is a strongly causal system, then S with inputs restricted to $[t_0,t_f)$ and outputs restricted to $[t_0,t_f]$ is a function and so is S with inputs restricted to $[t_0,t_f)$ and outputs restricted to $(t_0,t_f]$.

Property 4.3 $S[t_0,t_f]$ is causal if and only if $S[t_0,t_f]$ is a function.

Properties 4.1 and 4.2 state that special restrictions of a (strongly) causal system yield an I/O functional system. So, given a causal system, we can safely consider appropriate restrictions of it to be functions. Property 4.3 states that on an interval of finite width, every I/O functional system is a causal system and vice versa.

Property 4.2 demonstrates the predictability of strongly causal systems. Consider, for example, a simple delay function $y(t) = x(t-\tau)$. This system produces an output that is a copy of the input as it appeared τ units of time in the past. If we want to predict the

output of this system as some time t, it is sufficient to observe the input up to time t-τ. In a simulation system, we might take advantage of this by releasing future outputs based on current inputs (i.e., if we see an input event at time t, we can release an output event with time stamp t+τ). This observation translates into the concept of lookahead in conservative simulation algorithms developed for logical process models (see, e.g., [Zeigler 2000], [Fujimoto 2000]).

4.2. An I/O Functional Model of a Distributed Computing System

Now we turn to constructing an analog of the event sequence model in terms of state transition and coupled component models. Having done this, we redefine causal precedence in terms of input/output and component coupling functions. It is shown that the definition of causal precedence in these terms is a generalization of the causal precedence relation defined in terms of event sequences. The model from Chapter 3 is retained, in which a distributed computing system is treated as a set of processes that interact only via message passing, although an extension to it is necessary in order to construct a state transition representation.

Consider a single run of a distributed program. Each process in the run generates an execution trace $e_0e_1...e_n$ which is a finite sequence of local events. An event is some action that causes a change of state, for example, executing a procedure, sending or receiving a message, etc. Three types of events are considered significant in this model.

These are send events, receive events, and internal events. A send event occurs when a process sends a message to some other process. A receive event is the receipt of a send event. Internal events model autonomous behavior on the part of a process.

Event sequences can be seen as the input and output trajectories for a transition system (see [Tel 2000] for detailed treatment). A transition system is a state machine that has rules for processing receive events, internal events, and send events. In this model, receive events correspond to the consumption of a message and send events correspond to the production of a message. It is assumed that there is a set of all possible messages, denoted by M, which can be sent (via send events) and received (via receive events). This is made precise in Definition 4.2.

Definition 4.2 A transition system is a tuple $<C, I, \perp^r, \perp^s, \perp^i>$ where

C is a set of states,

I is a set of initial states,

$\perp^r \subseteq C \times M \times C$ is the transition function corresponding to a receive event for a message $m \in M$,

$\perp^s \subseteq C \times M \times C$ is the transition function corresponding to a send event for a message $m \in M$,

$\perp^i \subseteq C \times C$ is the transition function corresponding to an internal event.

The state transition behavior of this system can be described by a system

$<X, Y, S, \Delta, \Lambda>$

where

X and Y are sets of internal, send, and receive events,

S corresponds to the set of processor states C,

Δ = { (s,x,s') : x is an internal event and (s,s') $\in \perp^i$ or x is a receive event corresponding to a message m and (s,m,s') $\in \perp^r$ or x is a send event corresponding to a message m and (s,m,s') $\in \perp^s$ }, and

Λ = { (s,y) : y is the next internal, send, or receive event that the transition system intends to execute when it is in state s }.

The state transition function for this system is the union of state and event pairs occurring in the transition functions of the transition system. The output function describes the next event that the process will execute when it is in a particular state.

This definition of the output function agrees with the interpretation of an event sequence given in [Garg 1996]. An event sequence can be augmented to include the sequence of states that result from executing an event. So the sequence $e_0e_1...e_n$ becomes $s_0e_0s_1e_1...s_{n-1}e_n$ where s_i is a process state, e_{i-1} is the event that caused a transition from state s_{i-1} to s_i and e_i is the next action taken by the system when it is in state s_i. In our state transition representation of a process, this same behavior is obtained by using a feedback loop.

The input/output behavior of this system is a relation over sequences of events. The relation maps sequences of length N to sequences of length N+1. An input sequence

of length N corresponds to the sequence of events that caused the system to be placed into a particular state. The output sequence of length N+1 is prefixed by a copy of the input sequence and ends with the next event that the system wants to process. It is clear that the process is a strongly causal system since we only require the first N inputs to determine the value (N+1)st output.

When a distributed computation involves several processes, their joint behavior is modeled by another transition system which is constructed from the transitions systems that model each process.

Definition 4.3. Let **M** denote the collection of multisets with elements in the message set M. The transition system induced under asynchronous communication by a processes $p_1, p_2, ..., p_n$ is

$< C \times \mathbf{M}, I, \perp^r, \perp^s, \perp^i>$,

where each p_k is described by a transition system $<C_k, I_k, \perp^{rk}, \perp^{sk}, \perp^{ik}>$, and

$C = C_1 \times C_2 \times ... \times C_n$,

$I = I_1 \times I_2 \times ... \times I_n \times \{\varnothing\}$,

$\perp^i = \{ ((c_1, ... , c_k , ... , c_n),M_1),(c_1, ... , c_k' , ... , c_n),M_1)) \text{ such that } (c_k,c_k') \in \perp^{ik} \}$,

$\perp^s = \{ ((c_1, ... , c_k , ... , c_n),M_1),(c_1, ... , c_k' , ... , c_n),M_1 \cup \{m\}))$

such that $(c_k,m,c_k') \in \perp^{sk} \}$, and

$\perp^r = \{ ((c_1, ... , c_k , ... , c_n),M_1),(c_1, ... , c_k' , ... , c_n),M_1 - \{m\}))$

such that $(c_k,m,c_k') \in \perp^{rk}$ and $m \in M_1\}$.

For each message there is assumed to be a unique process, called the destination of the message, that can receive the message. Note that in this case, the transition system is defined not by transition functions but only by transition relations. This is because we compute the next state by executing a single event at a single process and we allow any order of events that result in equivalent executions [Tel 2000]. If events a and b are causally independent (i.e., a || b holds) then we can apply these events in any order and arrive at identical states. Consequently, the order in which the events are applied is irrelevant.

Consider a state transition representation of the message passing system. This system will accept as input send, receive, and internal events. The messaging system has one input/output port for each processor in the distributed system.

When the message passing system receives a send message from process p_k intended for process p_i the system stores a message for p_i and immediately produces a send event for p_k. When the message passing system is provided with a receive event from process p_k and there is a stored message for process p_k, then the message passing system removes the message from its storage and generates a receive event for p_k. If the message passing system receives an internal event from process p_k, then it immediately generates as output an internal event for process p_k.

Any other input (i.e., a receive event when no message is stored) causes the message passing system to generate a non-event, which is treated by the process as an internal event, and does not change the process state (e.g., it might correspond to waiting for a message). A working example of a message passing system such as the one

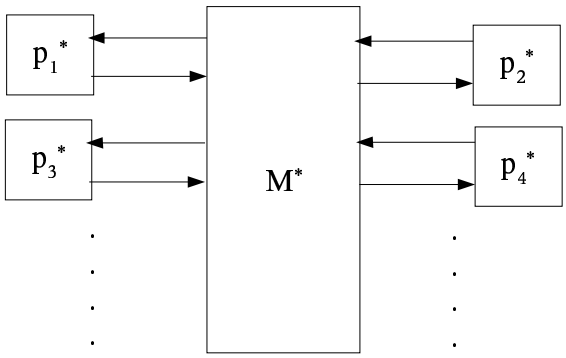

Figure 4.1. A multicomponent system consisting of several processes and a message passing system.

described can be found in the Message Passing Interface (MPI) standard (see [Snir 1998]). Note that the message passing system is a causal, but not strongly causal, system since it produces output immediately in response to an input.

Consider the multicomponent system depicted in Figure 4.1. The message passing system is denoted by M^* and the state transition representation of process p_k is denoted by p^*_k. If the original transition system includes non-events, then this model can be reduced to a smaller image of its transition system counterpart as follows.

This multicomponent model is reduced to a closed (i.e., input and output free) state transition model that is given by

$$<S, \Delta>$$

where

S is $C \times \mathbf{M}$ which is the state space of the associated transition system,

$$\Delta \subseteq \{ ((c_1, \ldots , c_k , \ldots . , c_n),M_1),(c_1' , \ldots , c_k' , \ldots . , c_n'),M_2)) \}$$

where for each component either

1) c' results from c when an internal event is applied, or

2) c' results from c when an send event is applied corresponding to a message m and m is in M_2, or

3) c' results from c when a receive event is applied corresponding to a message m and m is in M_1 and the number of instances of m in M_2 is exactly one less than the number of instances in M_1.

State trajectories for this new system coincide with a unique sequence of equivalent executions of the original transition system. The mapping from a transition system to a this new state space representation is demonstrated in the following example.

Example Consider two processes participating in a distributed computation. Each process executes an internal event. Next, the first process sends a message to the second process. This is followed by the second process sending a message to the first process. Each process can be modeled by a transition system

$C_1 = C_2 = \{ s_1, s_2, s_3, s_4 \}$,

$I_1 = I_2 = \{ s_1 \}$,

$M = \{ m_{12}, m_{21} \}$,

$\perp^r = \{ (s_3,m_{21},s_4) \}$ for the first process, and $\{ (s_2,m_{12},s_3) \}$ for the second process,

$\perp^s = \{ (s_2,m_{12},s_3) \}$ for the first process, and $\{ (s_3,m_{21},s_4) \}$ for the second process,

$\perp^i = \{ (s_1,s_2) \}$ for both processes.

Any other event applied to the transition system (including the 'non-event') does not change its state. The transition system induced by these processes under asynchronous communication is readily determined from Definition 4.3.

Starting from state (s_1,s_1), the induced transition system can proceed to state (s_2,s_2) in two different ways. An internal event can be applied to the first process and then to the second process or vice versa. From state (s_2,s_2) the system must proceed through states (s_3,s_2), (s_3,s_3), (s_3,s_4), and (s_4,s_4). So two possible (and equivalent) executions of the induced transition system are

$(s_1,s_1,\{\})(s_1,s_2,\{\})(s_2,s_2,\{\})(s_3,s_2,\{m_{12}\})(s_3,s_3,\{\})(s_3,s_4,\{m_{21}\})(s_4,s_4,\{\})$ and

$(s_1,s_1,\{\})(s_2,s_1,\{\})(s_2,s_2,\{\})(s_3,s_2,\{m_{12}\})(s_3,s_3,\{\})(s_3,s_4,\{m_{21}\})(s_4,s_4,\{\})$..

The corresponding sequence of states computed by the state space representation of this transition system is

$(s_1,s_1,\{\})(s_2,s_2,\{\})(s_3,s_2,\{m_{12}\})(s_3,s_3,\{\})(s_3,s_4,\{m_{21}\})(s_4,s_4,\{\})$.

Notice that the equivalent state sequences $(s_1,s_1,\{\})(s_1,s_2,\{\})(s_2,s_2,\{\})$ and $(s_1,s_1,\{\})(s_2,s_1,\{\})(s_2,s_2,\{\})$ have been replaced by the sequence $(s_1,s_1,\{\})(s_2,s_2,\{\})(s_3,s_2,\{m_{12}\})$. The latter sequence is the unique result that must follow from processing the causally independent internal events (see Theorem 2.21 in [Tel 2000]).

4.3. Causal Precedence in Multicomponent Systems

Another notion of causal precedence can now be defined in the context of multicomponent systems.

Definition 4.4 Let w, w', x, and y be abstract time functions and F a causal I/O functional system. A segment x immediately precedes segment y if

$$y\langle t_4,t_3\rangle w'\langle t_3,t_1\rangle = F(w\langle t_4,t_2\rangle x\langle t_2,t_1\rangle)$$

for some w, w', with F restricted to $\langle t_4,t_1\rangle$, and $t_1 \leq t_2 \leq t_3 \leq t_4$.

Causal precedence is defined to be the transitive closure of the immediately precedes relation. As before, this is denoted by $x\langle t_2,t_1\rangle \rightarrow y\langle t_4,t_3\rangle$.

It is possible to derive the classical (i.e., Definition 3.1) notion of causal precedence from Definition 4.4. Consider the restriction of a transition system with an I/O function F restricted to [k,k+m] and input sequences x and y. Then we have y[k+m, k+m]y(k,m] = F(x[k+m,m)x[m,m]). From Definition 4.4 event x(m) causally precedes y(k+m) and so we have arrived at condition (i) of Definition 3.1.

Now suppose that a process consumes a send event x(k). A message is then stored in the messaging system M^*, and at some later timer a receive event y(k+m) is ejected from M^*. This corresponds to x(k) being a send event and y(k+m) being the corresponding receive event. Let M^{**} be the I/O function for M^* and F the I/O function for the sending process. Let z_{MF} be a coupling function mapping M^* output segments to process input segments and define z_{FM} similarly. So we have

$$y[k+m,k+m]y(k+m,k] = z_{MF}(M^{**}(z_{FM}(F(x[k+m,k)x[k,k])))$$

and x(k) causally precedes y(k+m). We have arrived at condition (ii) of Definition 3.1. The transitive closure of these two cases completes the derivation.

4.4. Logical Processes

A 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 ([Bagrodia 1991],[Misra 1986]). Such a logical process is called realizable by [Misra 1986]. Clearly, a realizable logical process is a weakly causal system.

A predictable network of logical process is proposed by [Misra 1986] as a necessary condition for a logical process to be implemented on a parallel computer. A predictable logical process is one 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. A predictable network of logical processes is a network of logical processes in which each cycle contains at least one predictable logical process.

There are two aspects of the logical process world-view that introduce problems when trying to represent general systems models. The first is the requirement for a strongly causal logical process to exist in every simulation. This presents problems when trying to simulate collections of weakly causal systems. A violation of this principle can be seen in the implicit Euler approximation of $dx/dt=f(x(t))$ by the causal (but not strongly causal, so 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 networks of weakly causal DEVS models (see [Kofman 2002]).

The second difficulty is apparent from a careful study of the process model from which the logical process model is derived. The input trajectories are sequences of messages with exactly one message being processed at each point in time. When a model is intended to process many events simultaneously then we encounter the so called 'simultaneous event problem' (see, for example, [Rönngren 1999], [Wieland 1999], [Nicol 2000], [Barz 2003], and [Jha 2000]). Solutions to this problem have almost invariably relied on further information about the model itself (e.g., priorities), or are solved by fiat (e.g., first come, first served).

To illustrate the simultaneous event problem when simulating general systems, consider modeling an n-body gravitational problem using a collection of n logical processes. 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 modeling formalism becomes impractical. This observation has been made by several authors (e.g. [Frey 2002],[Ghosh 1996]) who have constructed correctness proofs for parallel simulation algorithms. One consequence of this impracticality is that existing correctness proofs restrict themselves to showing that messages are processed in time stamp order. The issue of correctness with respect to a modeling formalism (e.g., DEVS) is not addressed.

Both the strong causality requirement and simultaneous event problem appear to arise as a result of the local causality constraint. The local causality constraint is generally considered to be a fundamental starting point for reasoning about parallel discrete event simulation algorithms (see, e.g., [Fujimoto 2000] and [Ferscha 1995]).

> Definition 4.5. Local causality constraint. No causality error can occur in an asynchronous logical process simulation if, and only if, every logical process processes events in non-decreasing time stamp order.

Implicit in the local causality constraint is the assumption that the causal precedence relation over a set of events is contained in the time stamp order of those events. When simulating weakly causal systems, we can construct a counter example using Definition 4.4 as our definition of casual precedence. Let F be an I/O functional representation of a weakly causal system. Let x[t,t] be an input segment for the system and y[t,t] be the corresponding output segment. So we can write y[t,t] = F(x[t,t]) and so

the event x(t) causally precedes y(t). Since t = t this cannot be deduced from the times associated with y(t) and x(t).

4.5. Logical Clocks in Distributed Simulation

The detection of causal precedence relations over a set of events is crucial for constructing asynchronous distributed simulation algorithms. The use of simulation time stamps as a basis for doing this is inadequate when simulating more general systems. In fact, the type of logical clock that arises from this scheme is a minimally consistent logical clock [Rönngren 1999] .

When simulating weakly causal systems and multi-input systems, the ability to discern a causal relationship between two seemingly simultaneous events is necessary. If a weakly consistent clock could be found that is consistent with the definition of casual precedence given by Definition 4.5, then we would, in principle, be able to modify many existing parallel discrete event simulation algorithms to properly handle general systems.

A weakly consistent clock suitable for keeping time in simulations of weakly causal systems is presented in [Rönngren 1999]. This algorithm was originally introduced for the purpose of detecting simultaneous events and preventing causality errors in the case of zero time events when simulating logical process models. A time stamp is a pair (t,c) where t is a model derived time-stamp (i.e., the time associated with an 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 event time 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 new time. 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 $c_1 = c_2$.

Theorem 4.1 relates this logical clock to Definition 4.4 when we are concerned with discrete event systems.

Theorem 4.1. Let $x(t)$ and $y(t+\tau)$, $\tau \geq 0$ be two events in event trajectories x and y, which are restricted to a finite interval. Let C denote the above mentioned logical clock. If $x(t) \rightarrow y(t+\tau)$ then $C(x(t)) < C(y((t+\tau))$.

Proof. Let $C(x(t)) = (t,c)$. Suppose $\tau > 0$ and $y[t,t+\tau) = \Phi[t,t+\tau)$. In this case, we have $C(y(t+\tau)) = (t+\tau,0) > (t,c)$. If $\tau = 0$, then $C(y(t+\tau)) = (t,c+1) > (t,c)$. Now suppose there is a non-decreasing sequence $\tau_1, \tau_2, \ldots, \tau_n$ such that $y(t+\tau_i) \neq \Phi$ and y is equal to the non-event everywhere else. Moreover, suppose that $C(x(t)) < C(y((t+\tau_1)))$. By induction it can be concluded that $C(x(t)) < C(y((t+\tau_n)))$. ■

Using this clock, the time stamp order of events will contain the causal order of those events. The clock's operation can be imagined by drawing a directed, a-cyclic graph in which 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. A node with more than one incoming arc has several simultaneous inputs, with one input event corresponding to each arc. The labels are the numbers assigned to different zero time events by the second field of the simulation clock.

5. A FRAMEWORK FOR PARALLEL SIMULATION OF DEVS MODELS

This chapter presents a conceptual framework for mapping formal and semi-formal modeling constructs into a logical process world-view. The significance of this development is that it allows the task of model composition to be separated from that of simulator construction, and it enables the construction of simulation algorithms whose correct operation can be ensured (via a formal proof) for a class of systems. The utility of the framework is demonstrated via the development of a simple DEVS simulator.

5.1. A Framework for Distributed Simulation

A variety of architectural styles (e.g., Call-and-return and Independent Components [Bass 1997]) can be employed to synthesize a distributed simulation environment. Among various candidate architectural styles, we have chosen a layered style in order to separate distinct functionalities that are needed for representing and executing model components and simulation protocols in distributed settings [Sarjoughian 2000][Sarjoughian 2001]. The layered modeling and simulation architecture consists of modeling, simulation, and middleware layers. These layers provide a basis to integrate different types of software components and further analyze key features of modeling and simulation paradigms. The modeling layer components can represent (homogenous or heterogeneous) model specifications. Components in the simulation layer represent (sequential, parallel, or distributed) simulation protocols that

correspond to the models specified in the modeling layer. The middleware layer consists of software components that provide services (e.g., causal ordering of multiple messages) that are necessary for the simulation protocols to correctly interoperate. Figure 5.1 shows the organization of a simulation framework for a general modeling formalism.

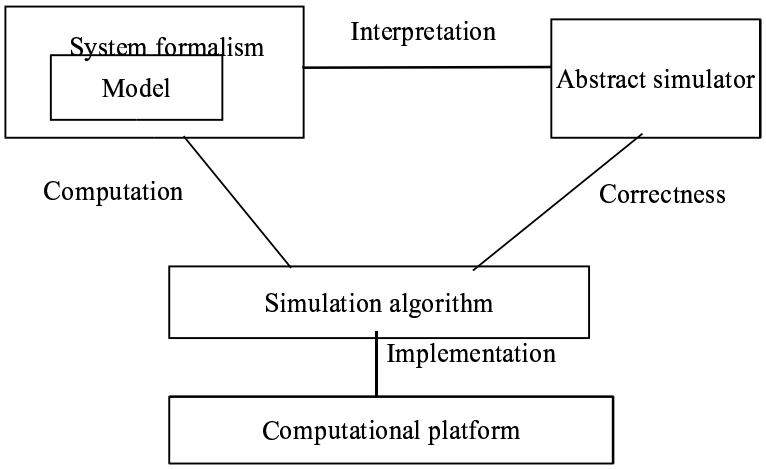

Figure 5.1. The elements of a modeling and simulation framework.

An architectural framework facilitates studying simulation protocols and distributed computational techniques by having an abstract view of the modeling formalisms under consideration. This abstraction is useful because it describes the formalism in a computationally friendly way. For example, in this work 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. A particular choice of clock (i.e. weakly or minimally consistent) describes the relationship between time and causality in the modeling formalism. The two dimensional clock described in section 4.5 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.

As shown in Figure 5.1, 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, presented in what follows, 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 discrete event system is a structure that is equivalent to the system at the I/O functional level, and whose time stamped events form a partially ordered set that corresponds to causal chains as defined by the causal precedence relation. 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. $e_1 \leq e_2$ iff $t_1 \leq t_2$

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 •. For example, let (t,a) and (t,b) be two causally independent events. Then (t,a)•(t,b) ≡ (t, {a,b}). Concatenation allows bags of simultaneous events 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 is a stack of events that records the input 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 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 $e_n < y \leq eN(e_0e_1...e_n)$.

If a minimally consistent clock is assumed, then

$eN(e_0e_1...e_n) = e$ such that $e_n \leq e$, and

$out(e_0e_1...e_n) = y$ such that $e_n \leq y \leq eN(e_0e_1...e_n)$.

5.2. Computational Representation of Logical Processes

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 logical process, the operation push(s,x) constructs the event

history that results from processing x. The out(s) function is used to determine what 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 [Ghosh 1996]) and SPEEDES which allows for it (see [Steinman 1992]).

5.3. 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 demonstrated in Algorithm 5.1.

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 [Zeigler 1999]. 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.

5.4. An Event Stepped 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 [Zeigler 2000]). An inductive argument is used to demonstrate the correctness of the algorithm with respect to the DEVS abstract simulator.

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., $x \bullet \Phi = x$). It is assumed that the clock is weakly consistent. The "for each" blocks are executed in parallel.

Algorithm 5.1 (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
```

Consider the jth simulator. Let s_j be the correct event sequence for that simulator. If the event sequence is empty, then the algorithm is trivially correct since it processes no events and produces no output.

Now, suppose the algorithm has proceeded without error and, for each simulator, has produced at line 2 the correct event sequence s_i which is a prefix of the final correct event sequence. Consider the next pass of the algorithm at the jth simulator. The next event can be added as a result of an internal transition, external transition, or a confluent transition or no event takes place.

Assume that no correct event can be processed in the next pass. From the induction hypothesis, at line 18, tN is found to be less that $eN_j(s_j).t$. So at line 7 the next event is set to the null-event. Line 12 is not executed since that would require that some influencer of j processed an incorrect event in the previous iteration of the algorithm, contradicting the induction hypothesis. So line 15 evaluates to false a no event is pushed onto the stack.

Assume the next correct event is an internal event. At line 18, tN is set to $eN_j(s_j)$. So line 4 evaluates to true. Line 12 is not executed since this would require that some influencer of j processed an incorrect event, contradicting the induction hypothesis. So at line 16, $eN_j(s_j)$ is pushed onto the stack.

Now, assume the next simulation event for j is an external transition. At line 18 on the previous pass, tN is computed to be less than $eN_j(s_j)$. So line 4 evaluates to false. Since the s_i are assumed to be correct, then line 12 is executed for all and only the correct influencers of j, thereby constructing the next correct event to be pushed onto x_j. At line16 the event is processed.

Finally, assume the next simulation event for j is a confluent transition. At line 18 on the previous pass, tN is computed to be equal to $eN_j(s_j)$. So line 4 is evaluated. Since the s_i are assumed to be correct, line 12 is executed for all and only the correct influencers of j, thereby constructing the next correct event to be pushed onto x_j. At line16 the event is processed.

Additional properties that we would like to show are liveness (freedom from deadlock), and correct termination (the simulation halts when the desired stacks have been created). 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}.

In the next two chapters we will employ this framework to prove correctness of optimistic and semi-optimistic parallel simulation algorithms. The proofs will be

considerably more demanding than the one just given and the utility of the framework in constructing the proofs will become clear.

6. OPTIMISTIC SIMULATION OF DEVS MODELS

The time warp algorithm has been used extensively to implement fast, parallel discrete event simulations, and has been developed in terms of a logical process world-view (see, e.g., [Lowry 2000],[Carothers 2000],[Steinman 1992],[Mascarenhas 1995], [Deelman 1996], and [Martin 1996], the concept first appeared in [Jefferson 1985]). A time warp algorithm for simulating DEVS models was developed in [Christensen 1990]. Several years of active research has produced a collection of refined and highly tuned time warp variants for solving problems posed as a network of communicating logical processes.

In this chapter, a variant of the time warp algorithm is constructed for simulating DEVS models, and it is unique in two respects. First, it properly handles instantaneous functions and the confluent transition function. Second, it introduces a novel form of message cancellation using implicit anti-messages. Potentially, this can improve the performance by reducing the amount of network traffic generated during a rollback (i.e., avoiding anti-message explosions). It also reduces the memory requirements as we no longer need to retain an output history for each process.

Implicit anti-messages are used to cancel output segments generated by DEVS models. In contrast to this, the explicit anti-messages used in aggressive and lazy cancellation schemes undo the effects of individual events within a segment (see, e.g., [Rajan 1995], [Fujimoto 2000], and [Ferscha 1995]). For example, consider the event segment shown in Figure 6.1. Suppose the process that generated the output segment

receives an anti-message with time stamp $t_{rollback}$ at the dashed line. 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 to $t_{rollback}$. The implicit anti-message scheme will send one anti-message which cancels all events in the interval $[t_{rollback},\infty)$. As the number of events in that interval increases, the magnitude of the reduction in anti-messages, with respect to the aggressive and lazy schemes, is expected to increase as well.

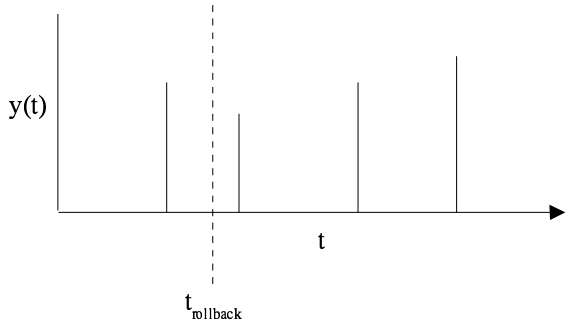

Figure 6.1. An output segment for a discrete event model.

To support rollbacks, a time warp algorithm must construct and store a series of checkpoints by periodically recording the process state and its input/output history. Inevitably, a time warp system will exhaust all available memory unless a scheme is in place for finding and deleting checkpoints that are no longer useful. This is called fossil collection.

Fossil collection in a time warp system has been an active area of research (see, e.g., [Soliman 99], [Christopher 1998], [Bruce 1995]). Here we describe a simple fossil collection scheme that illustrates the basic concepts. First, find a consistent cut. Next,

find the minimum event time stamp in the future partition of the cut (i.e., find an approximation of the global virtual time). Each process retains exactly one checkpoint whose time stamp is less than the global virtual time approximation. All other checkpoints whose time stamps are less than the global virtual time are destroyed. The retained checkpoint is used to ensure that we can roll each process back to some consistent state (i.e., a state from which no further rollbacks would be required).

6.1. A Time Warp Algorithm for DEVS Models

Suppose we have a set of processors labeled 1, 2, 3, ..., N. Let P denote the set of process labels. Processes communicate by sending messages which are pairs (m,e) where e is an event and m is the label of the sending process. *All events carry time stamps generated by the logical clock described in Section 4.5.* This is required for the algorithm to correctly handle simultaneous events. For any process k, we maintain a message bag A_k that contains the messages available at process k. There is also a message bag U_k that contains the messages whose events have been used by process k. Each process k also has a set of event sequences S_k that is the state history of process k.

Given an event sequence $s \equiv e_1e_2...e_n$, we define s.t to be the time stamp associated with event e_n. If the sequence s is empty, s.t is defined to be (0,0). The min operator over A_k, U_k, and S_k is defined as follows: min A_k gives the smallest event time stamp in A_k. If A_k is empty, define min $A_k = (-\infty,-\infty)$. Similarly, max A_k is the largest event time stamp

in A_k. If A_k is empty, define max $A_k = (\infty,\infty)$. The operator min over U_k is defined in exactly the same way. Finally, min S_k gives the smallest s.t over all $s \in S_k$.

The symbol Φ is used to denote 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 special value that means 'rollback to time t', where t is the event time stamp.

First in-first out (FIFO) channels are assumed for inter-process communication. The command **receive** returns the next message available to the process, or the special symbol **blank** if no message is available. The command **send**(m) sends message m to the appropriate processes. The appropriate processes are defined by the system coupling, which is not considered here explicitly. Rather, it is assumed that some fixed system coupling is known prior to beginning a run (i.e., **send**(m) always sends the message m to the same set of processes)[1]. The **send** and **receive** command are a typical model of a FIFO communication channel. Implementations of this model are available in software packages such as the Message Passing Interface (MPI, see [Snir 1998], also see [Wolfgang 1994] and [Hwang 1993] for a discussion of parallel computer programming models).

The time warp algorithm is listed as Algorithm 6.1. The variable s is an event sequence. The sets A_k, U_k, and S_k are as defined above. The algorithm is run at each process involved in the computation. The statement m.event, where m is a message,

1. The **send**(m) command actually imposes a stricter requirement than is absolutely necessary. A weaker condition that requires **send**(m) to transmit rollback messages only to processes that previously received a non-rollback message from the sending process is sufficient.

refers to the event associated with m. For brevity, we use m.t to refer to the time stamp assigned to m.event. Similarly, for an event e, we use e.t to refer to the time stamp assigned to event e. The statement m.proc is used to refer to the process label associated with the message m.

Algorithm 6.1 (Time Warp simulation)

1. $s := \Phi$
2. $A_k := \Phi$
3. $U_k := \Phi$
4. $S_k := \{s\}$
5. while (terminating condition not met)
6. msg := **receive**
7. if (msg $\neq$ **blank**)
8. if (msg.event = **r**)
9. $A_k := A_k - \{(m,x)$ s.t. m = msg.proc and msg.t $\leq$ x.t and $(m,x) \in A_k\}$
10. $U_k := U_k - \{(m,x)$ s.t. m = msg.proc and msg.t $\leq$ x.t and $(m,x) \in U_k\}$
11. end
12. if (msg.t $\leq$ s.t)
13. $S_k := S_k - \{z$ s.t. $z \in S_k$ and z.t $\geq$ msg.t$\}$
14. $T := \{(m,x)$ s.t. x.t > (max S_k).t and $(m,x) \in U_k\}$
15. $U_k := U_k - T$
16. s := max S_k
17. $A_k := A_k + T$
18. **send**(**r**) with **r**.t = msg.t
19. end
20. if (msg.event $\neq$ **r**)
21. $A_k := A_k + \{msg\}$
22. end
23. end
24. $A := \{a_1,a_2,...a_n$ s.t. $a_i \in A_k$ and a.t = min $A_k\}$
25. $x := a_1 \bullet a_2 \bullet ... \bullet a_n$
26. if (x.t = eN(s).t)
27. send(out(s))
28. $A_k := A_k - A$
29. s := push(s,x•eN(s))
30. $U_k := U_k + A$
31. else if (x.t < eN(s).t)

```
32.             A_k := A_k -  A
33.             s := push(s,x)
34.             U_k := U_k + A
35.     else
36.             send(out(s))
37.             s := push(s,eN(s))
38.     end
39.     S_k := S_k ∪ {s}
40.end
```

The algorithm works as follows. At the beginning of each iteration, we check for any available messages. If a message is available, we first check to see if it is a rollback indicator. If so, we remove from our list of available inputs all available messages received from the sender. This has the effect of canceling incorrect messages that have been received from the sender but not yet processed. Next, we check to see if the received message is in our simulated future. If so, we simply add the message to the bag of messages available to be processed. Otherwise, we restore the model to a checkpoint state just prior to the message time stamp. This includes restoration of the model state, the available input message bag, and the used message bag. A rollback notification is sent to the processes' neighbors to inform them that outputs with a time stamp greater than or equal to the rollback time should be discarded (i.e., they are incorrect). Lastly, we add the received message to the bag of available messages and then proceed as before.

Lines 23 through 37 compute the next state of the DEVS model. To do this, we check the available inputs and, if input is available, construct a single external event by concatenating the available events whose time stamp is equal to min A_k. If this time stamp is equal to the model's time of next event, we send the model's output, compute a

confluent transition function, and move the processed inputs to the used event bag. If the input time stamp is less than the model's time of next event, we compute an external transition function and copy the processed inputs to the used event bag. Finally, if the available input is larger than the model's time of next event, we send the model output and compute an internal event. At line 37, the model state is saved and we proceed to the next iteration.

The correctness proof for this algorithm is lengthy, and it consists of two halves. The first is a safety proof that shows the algorithm is correct up to global virtual time. That is, if global virtual time has advanced to time (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 that shows global virtual time increases.

Figure 6.2 provides roadmap for the correctness proof. The bubbles indicate significant steps in the proof. Logical dependencies between steps are shown by solid lines connecting the bubbles, with bubbles near the top of the the picture preceding bubbles that are closer to the bottom. The bubbles that have no logical predecessors are proven directly from the program text and using facts established in previous chapters. The last step in the safety proof is to demonstrate that the correct event property holds. The proof of a non-decreasing GVT and always eventually increasing GVT is left for the liveness proof. It will follow immediately from the liveness and correct event theorems that the algorithm correctly simulates DEVS models.

The safety proof is constructed as follows. First, we will show that there is always a checkpoint sufficiently far in the past to allow for rollbacks. An obvious proof

might already be apparent to the reader. We will opt for a more complex proof that allows for fossil collection by assuming only that a suitable checkpoint existed in the previous iteration of the algorithm.

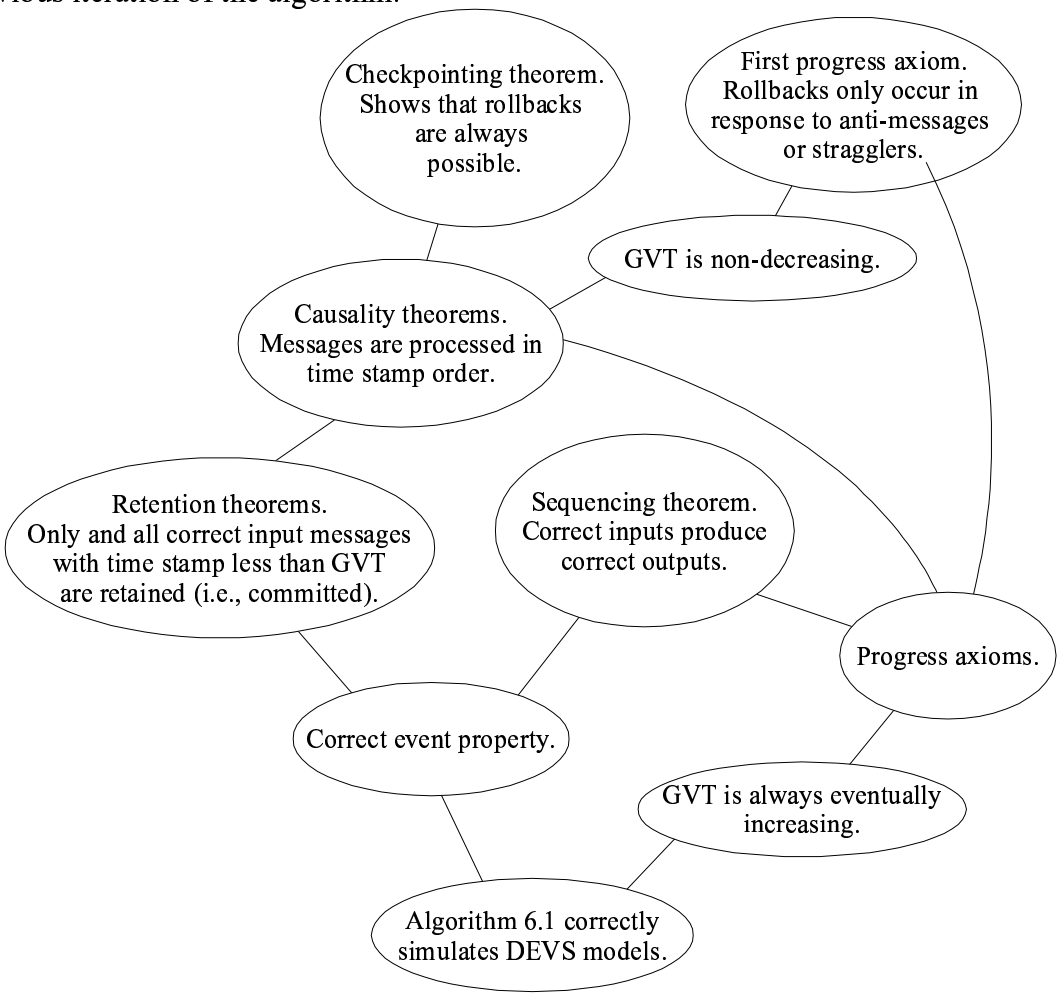

Figure 6.2. A road map for the Time Warp algorithm correctness proof.

Having proved that rollbacks can always be computed, we will then prove a pair of causality theorems which, taken together, state that events are processed in time stamp order. This property, in conjunction with the checkpointing theorem, are needed to prove a pair of retention theorems. The retention theorems state that all correct input events are

retained and no incorrect input events are retained up to global virtual time. The sequencing theorem is proven next, and it states that the algorithm produces correct output given correct input. The conclusion that the algorithm is correct up to global virtual time follows immediately from the preceding theorems.

6.2. Checkpointing Theorem

Informally, the checkpointing theorem states that it is always possible to perform the rollback step at line 15. This is done by showing that S_k always contains a sequence s such that s.t is less than the smallest time stamp than can be received from another process. By definition, the time stamp of any event received by a process is greater than or equal to GVT. So we need only be careful that the algorithm keeps at least one checkpoint whose time stamp is less than GVT.

Theorem 6.1 (Checkpointing Theorem) If GVT is non-decreasing, then min $S_k <$ GVT and $S_k \neq \Phi$.

A simple proof can be constructed.

Proof 1 Initially, $S_k = \{q_k\}$, q_k being the initial sequence at process k, and $q_k.t = (0,0)$. It follows that GVT must be larger than (0,0). For any process k, the sequence s_k at the process must be such that $(0,0) \leq s_k.t$. Since $out(s_k).t > s_k.t$, $out(s_k).t > (0,0)$. A process can send either $out(s_k)$ or a rollback event **r**. Since $out(s_k).t > (0,0)$, at line 13 we never remove the initial state q_k from S_k in response to a message from a process p carrying $out(s_p)$ since $out(s_p) > (0,0)$. At line 18, note that no process will ever send a

rollback message with time stamp (0,0), so no process can receive a rollback message r with time stamp (0,0). Consequently, the initial state q_k will never be removed from S_k. ■

If we introduce a fossil collection system, this proof is no longer appropriate since it relies on the algorithm holding the initial checkpoint forever. A somewhat more complicated proof allows for correctness so long as a fossil collection scheme similar to the one described above is used.

Proof 2 Prior to executing the first iteration of the algorithm, the theorem holds as determined in proof 1. Now, suppose the theorem holds at line 5 after an arbitrary number of iterations. Consider the next iteration of the algorithm. Since GVT is assumed to be non-decreasing, if S_k is unchanged then the theorem continues to hold. It is sufficient to consider only those lines where S_k is altered.

At line 13, we remove from S_k only those elements $p \in S_k$ such that p.t is greater than or equal to the time stamp of the received event. The received message is an unprocessed event and so has a time stamp greater than or equal to GVT. By the induction hypothesis, min S_k < GVT. Clearly min S_k can not decrease as a result of executing line 13. So min S_k < GVT must still hold after executing line 13. Also, prior to executing line 13, the induction hypothesis requires at least one event $q \in S_k$ such that q.t < GVT. This element is not removed when line 13 is executed, so S_k contains at least one element with time stamp less than GVT after executing line 13.

At line 39, a sequence s is added to S_k 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. From the induction hypothesis, we know that min S_k < GVT prior to executing line 39. Clearly, min S_k can not increase as a result of executing line 39. So min S_k < GVT must still hold after executing line 39. Applying the same reasoning as in the previous case, it can be shown that S_k must contain at least one element with time stamp less than GVT after executing line 39. At no other point in an iteration is S_k altered. ■

6.3. Causality Theorems

There are two causality theorems. The first states that, if any events are marked as processed (i.e., that are in the used event bag), these events have time stamps in the past. The second causality theorem states that all unprocessed events (i.e., that are in the available event bag) are in the future.

Taken together, the causality theorems state that events are processed in time stamp order. It should be clear that this is a necessary, but not sufficient, condition for correctness. For instance, a simple way of meeting this criteria is to toss out any events that would cause out of order processing. An extreme solution is to discard all events! If no events are processed, then none of them are processed out of order. Clearly, this is not a correct simulation. Instead, we will require the retention theorems and a sequencing theorem to complete the proof. These will be considered shortly.

Theorem 6.2 (First Causality Theorem) Suppose GVT is non-decreasing. If $U_k \neq \Phi$, then max $U_k \leq$ s.t.

Proof Let e be the largest event in U_k. Then e must have been added to U_k at lines 30 or 34. In both cases, $e \in A_k$ was concatenated with zero or more events with identical time stamps (and possibly with eN(s)) and then pushed on to s. So it must be that e.t = s.t after executing line 30 or line 34. Since e is the largest element in U_k, max $U_k \leq$ s.t.

At lines 10 and 15, we only remove elements from U_k. So either max U_k stays the same, is reduced, or U_k becomes empty. If this is our first removal of elements from U_k, the theorem must have held before executing line 10 or 15. Since s.t remains the same and max U_k does not grow or U_k becomes empty, the theorem must remain true after executing lines 10 or 15. It follows that if we only alter U_k by executing lines 30, 34, 10, and/or 15, the theorem will always holds so long as s.t is not altered.

Increasing s.t without altering U_k, as is done at line 37, will not effect the truth of the theorem. This leaves line 16 where s.t is, possibly, reduced. At line 15, we remove from U_k all messages with time stamps larger than (max S_k).t. At line 16, we set s to max S_k. So it must be that max $U_k \leq$ (max S_k).t = s.t so $U_k \leq$ s.t. ■

Theorem 6.3 (Second Causality Theorem) Suppose GVT is non-decreasing. If $A_k \neq \Phi$, then min $A_k >$ s.t.

Proof Consider the first iteration of the algorithm. Initially, A_k is empty. The first message added to A_k must occur at lines 17 or 21. If msg.t > s.t, line 17 is not executed and at line 21 we add msg to A_k. Since this is the only message in A_k, clearly

min A_k > s.t. If mst.t ≤ s.t, then line 17 is executed. At line 17, the elements of T, all of which are larger than s.t, are added to A_k. So A_k is empty or min A_k > s.t after executing line 17.

The last possibility is that msg is **blank**, in which case no message is added to A_k. In any event, the theorem holds after reaching line 24. 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_k is greater than or equal to s.t. Suppose it is not, then a rollback must have occurred at line 12. After the rollback, s.t < msg.t. If line 20 evaluates to true, then s.t ≤ msg.t = min A_k. This contradicts the assumption that min A_k ≤ s.t. Otherwise, A_k is empty and the theorem is vacuously true.

Having established the truth of the theorem on the first iteration, suppose it holds at line 5 after an arbitrary number of iterations. Consider the next iteration. At line 6, we receive a message msg or **blank**. If **blank**, the theorem must still hold when we reach line 24 since neither A_k nor s are altered.

Now, suppose msg.event = **r**. Executing line 9 either makes A_k an empty bag or min A_k 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 12, the theorem still holds.

Next, suppose that line 12 evaluates to true. At line 16, we set s.t < msg.t At line 17, we add messages with time stamps larger than s.t. So the theorem still holds at line 20.

Finally, at line 21 we can add a message to A_k whose value is larger than s.t. If this was not the case, line 12 would evaluate to true causing s.t to be set back to a value less than msg.t, which contradicts the assumption that msg.t ≤ s.t.

Having safely reached line 24, we need only show that x.t is greater than s.t at line 25. It has been established that min A_k is greater than s.t. Note that x is the result of concatenating the events in A_k with time stamps equal to min A_k. It follows that x.t = min A_k. So s.t < x.t as desired. ■

6.4. Retention Theorems

While the causality theorems show that events are processed in time stamp order, the retention theorems state that all correct events with time stamp less than GVT have been processed, and no incorrect events with time stamp less than GVT have been processed. Taken with the causality theorems, this means that the correct events have been processed and they have been processed in the right order. It remains to be shown 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 6.4 (First retention Theorem) Select an arbitrary time T < GVT and assume that GVT is non-decreasing. Then the bag A_k contains no events with time stamps less than or equal to T.

Proof If A_k is empty, the theorem is obviously true. Otherwise, we know that min A_k > s.t (Theorem 6.3). Now, either s.t > GVT or s.t < GVT (s.t = GVT is ruled out by

the definition of GVT). If s.t > GVT the theorem is true. If s.t < GVT, then either A_k contains the smallest unprocessed event in the system and so min A_k = GVT or it is the case that min A_k > GVT. In either case the theorem holds. ■

Theorem 6.5 (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_k) 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_k 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 number of events processed and messages received system-wide. The theorem is clearly true when no events have been processed and no messages have been sent. 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 results in a rollback, then only events with time stamps greater than or equal to GVT will be removed from U_k at lines 10 and 15. Consequently, the theorem continues to hold when we reach line 23.

Now, suppose that the next activity to occur is the processing of an event. The event can have a time stamp greater than or equal to GVT (Theorem 6.4). If the time stamp of the event is greater than GVT or if the event is an internal 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 event is correct and the theorem holds in this case as well. That the event is correct follows from the induction hypothesis since 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. ■

6.5. 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 Algorithm 6.1.

For the moment, assume we have a bag of correct, available inputs. Let A be this bag and consider the algorithm with lines 6 to 23 removed. That is, assume we know what inputs need to be processed and that our job is just to process them correctly. Then the sequencing theorem is as follows.

Theorem 6.6 (Sequencing Theorem) Consider the algorithm with lines 6 to 23 removed and the bag A_k filled with all and only correct inputs up to time T. Then the algorithm produces only correct outputs up to time T.

Proof See Theorem 2.1 and the discussion in Section 5.4. Line 29 corresponds to a confluent event. Line 33 corresponds to an external event. Line 37 corresponds to an internal event. Lines 27 and 36 produce outputs. ■

The sequencing theorem justifies the assumption made in the second retention theorem concerning correct outputs. This leads to the correct event property that was our original goal.

Theorem 6.7 (Correct event property for Algorithm 6.1) Algorithm 6.1 is correct up to time $T < GVT$.

Proof Theorem 6.7 is an immediate consequence of the sequencing theorem and second retention theorem. ■

Theorem 6.7 states that the algorithm will generate the input and output trajectories that are correct for a given model, with respect to the DEVS formalism. The safety theorem was built up from three parts. First, events must be processed in time stamp order. Second, only and all correct event must be processed. Third, those events are interpreted correctly with respect to the DEVS formalism (i.e., as internal, external, and confluent events).

6.6. Liveness Theorems

The liveness proof will be taken almost directly from [Leivent 1993], where it was shown that if certain progress axioms hold, then global virtual time always eventually increases. The liveness theorem is needed to satisfy the hypothesis of Theorem 6.7 whic states that global virtual time is non-decreasing. Sufficient conditions for global virtual time to be non-decreasing are described in [Leivent 1993]. It is

sufficient then to show that the progress axioms stated by [Leivent 1993] hold for Algorithm 6.1. 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 whose time stamp 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 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.

It is easy to see that the output function requirement is exactly met by our simulation process model if a weakly consistent clock is used. That progress axiom 1 holds is clear from the algorithm text (see lines 8 and 12). To see that progress axiom 2 holds requires that internal events be treated as messages sent by a process to itself. Line 20 in conjunction with the causality theorems ensures that progress axiom 2 holds. The sequencing theorem ensures that progress axiom 3 holds (here, of course, we must consider the event $x = a_1 \bullet a_2 \bullet \ldots \bullet a_n$ to be under consideration).

Lemma 6.1 states that GVT is non-decreasing. [Leivent 1993] states the lemma in terms of a computation performed by a computational system. For our purposes, we only consider the properties of the computational system itself. Lemma 6.1, appropriately paraphrased, is as follows:

Lemma 6.1 [Leivent 1993] If a computational system satisfies progress axiom 1, then GVT in a computation performed by the computational system is non-decreasing.

Lemma 6.1 satisfies the hypothesis of the correct event theorem (i.e., GVT is non-decreasing). To show that actual forward progress is inevitable, we rely on Theorem 6.9 [Leivent 1993]. First we need a few definitions.

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 Algorithm 6.1 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 6.9 [Leivent 1993] Assume an error-free computation 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 6.9 shows that Algorithm 6.1 will reach a state in which all of the models are passive or, given any desired simulation end time, the algorithm will reach it.

6.7. Correct Simulation of DEVS Models

From Chapter 2, we know that the time warp algorithm correctly simulates a DEVS model if the algorithm exhibits the correct event property and simulation time always eventually progresses. Both of these properties have been shown to hold for Algorithm 6.1 and so we have

Theorem 6.10. Algorithm 6.1 correctly simulates the class of DEVS models.

Proof. Follows immediately from Theorem 6.7 and Theorem 6.9. ■

Algorithm 6.1 and its correctness proof have demonstrated all of the concepts developed in the previous chapters. The weakly consistent clock developed in Chapter 4.5 has been used to enable the correct handling of simultaneous and instantaneous events. This enables the optimistic algorithm to correctly simulate models that are weakly causal and have multiple inputs.

The algorithm uses a simple, but efficient, event cancellation mechanism built around the event trajectories used in the systems oriented modeling framework. The advantage of this approach can be seen in the relatively simple proofs of the causality theorems and the absence of complex event cancellation code in the algorithm itself.

The correctness proof itself relies on a clear separation of the model and its simulator. The correctness proof only states that, given a model expressed as a DEVS, Algorithm 6.1 will produce the same I/O 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 legitimate DEVS). The algorithm will generate correct behavior for any DEVS model, but in no way guarantees or requires that the model itself be 'correct' (i.e., valid).

7. RISK-FREE OPTIMISTIC SIMULATION OF DEVS MODELS

Risk-free optimistic algorithms have been proposed as a solution to the anti-message explosion problem that can appear in time warp simulations (see, e.g., [Dickens 1990], [Steinman 1992], and [Damitio 1994]). This class of algorithms has also been discussed by [Liao 1993] in connection with simulating sparse DEVS models. For simulating DEVS approximations of continuous systems, risk-free optimistic algorithms become attractive, not as a means to solve the anti-message problem, but to avoid rollback thrashing that can occur when neighboring logical processes have similar time advance values (see, e.g., [Ferscha 1999], [Das 2000]). This happens frequently when simulating continuous systems with smooth derivatives, and so thrashing by the time warp algorithm can be a significant impediment to efficient simulation.

To see how thrashing can occur, consider the situation depicted in Figure 7.1. Process A sends a message to process B with time stamp t. While this message is in transit, process B sends a message to process A with time stamps $t' < t$. Prior to receiving either of these messages, both process A and process B manage to execute at least one more speculative state transition. When the message with time stamp t' is received by process A, it causes a rollback to some time prior to t'. The resulting anti-message forces B to rollback to some time prior to t. In this example, we have computed one valid event (i.e., the output event sent by process B to process A with time stamp t') at the cost of two or more optimistic events and two rollbacks.

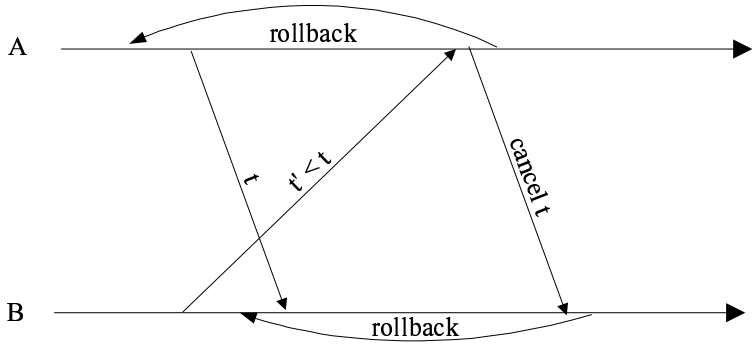

Figure 7.1. Thrashing in time warp when two processes have similar time advance values.

When state and output trajectories of a continuous system change smoothly in both time and space, neighboring cells residing on different processors will tend to have similar time advance values. This results in frequent thrashing when derivatives at process boundary cells are not zero.

One way to prevent the thrashing problem is to avoid exchanging speculative outputs. When events must be communicated across process boundaries, we can ensure that this is done in an orderly fashion by computing the global virtual time and allowing only the process whose output is correct (i.e., whose time of next event is equal to the global virtual time) to send its output. So long as interprocess events are infrequent, the risk-free optimistic algorithm can process internal events in parallel while searching for the first output that indicates a need to communicate with a neighboring process. For this purpose we can adopt the risk free optimistic algorithm presented in [Steinman 1992]. Similar to the time warp algorithm developed in Chapter 6, the new risk free optimistic algorithm correctly handles the confluent transition function and sequences of instantaneous events.

7.1. A Risk Free Optimistic Algorithm

Suppose we have a set of processors labeled 1, 2, 3, ..., N. Processes communicate by sending messages (i.e., no shared memory is assumed). Event time stamps are generated using the logical clock described in Section 4.5. Each process k has a set of event sequences S_k that is the state history of process k. Given an event sequence $s \equiv e_1 e_2 ... e_n$, we define s.t to be the time stamp associated with event e_n. The element in S_k whose s.t value is largest is denoted by max S_k. If the sequence s is empty, s.t is defined to be (0,0). The symbol Φ is used to denote an empty bag, set, or sequence. The meaning should be clear from the context in which the symbol is used.

Two distributed operations are assumed to be available. First, we expect to have a reduction operation min(t) which computes a global minimum over the values of t supplied by all processes. Secondly, we assume there is a distributed operation exchange_io(y), where y is an event value, that returns an event which is constructed by concatenating the supplied y's according to the model coupling. One implementation of the global minimum calculation is described as Algorithm 3.2. The exchange_io(y) function can be implemented by having each processor send its messages and then finding a consistent cut using Algorithm 3.1. The values appearing in the cut are returned to the calling process.

The basic elements of the risk free optimistic algorithm are listed as Algorithm 7.1. The variable s is an event sequence. The set S_k is as defined above. The algorithm is run at each process involved in the computation. For an event e, we use e.t to refer to the time stamp assigned to the event and e.v to refer to the value assigned to the event.

Algorithm 7.1

```
1.  s := Φ
2.  Sk := {s}
3.  while (terminating condition not met)
4.        while(out(s).v = Φ)
5.              s := push(s,eN(s))
6.              Sk := Sk ∪ {s}
7.        end
8.        GVT := min(eN(s).t)
9.        if (GVT = eN(s).t)
10.             y := out(s).v
11.       else
12.             y := Φ
13.       end
14.       x := exchange_io(y)
15.       if (x ≠ Φ)
16.             Sk := Sk - { s | s.t ≥ x.t and s ∈ Sk }
17.             s := max Sk
18.             if (GVT = eN(s).t)
19.                   s := push(s,(x,GVT)•eN(s))
20.             else
21.                   s := push(s,(x,GVT))
22.             end
23.             Sk := Sk ∪ {s}
24.       end
25.end
```

The algorithm operates in cycles, where the end of each cycle marks the end of a sequence of internal events at each processor. The search for an output at each processor occurs in the "while" loop at lines 4 through 7. Given a correct state prior to beginning

the loop, the loop computes a sequence of internal events and terminates when an output is found. The state of the model is recorded at line 6 following each change of state. Note that time is strictly increasing if we are using a weakly consistent clock and so the max S_k operation is well defined and any output event must have a value greater than (0,0).

Upon finding an output, the process participates in a global minimum calculation. Each process supplies its output time to the minimum operation. The smallest of these times is the global time of next event. Processes with output at that time have found correct output events, assuming that we began the loop at lines 4 through 7 in a correct state.

Those processes with correct outputs transmit them to their proper destinations at lines 9 through 14. Only processes with correct output events are allowed to send their outputs and all such processes do this. Consequently, the receiving processes will receive all and only correct input events at the global virtual time.

Those processes that receive inputs at the global virtual time will consume those input events at lines 15 through 24. At line 16, all incorrect states that have been recorded are removed from the set of recorded states. That is, all states that could have been causally preceded by the input event or that may have been concurrent with it are removed. Assuming the use of a weakly consistent clock, this is accomplished by removing all stored states whose last event time is greater than or equal to the input event time. Next, the model is restored to the recorded state whose last event time is nearest to the input event. That this state is the correct state follows from the computation of

correct internal events during the output search at lines 4 through 7, and the fact that the correct input trajectory in the interval [(max S_k).t,x.t) is null valued.

Having found the correct state preceding the input event, the input event is processed as either an external event at line 21 or as a confluent event at line 19. The resulting state is recorded in order to maintain an unbroken record of the state trajectory. This marks the end of a cycle. At the end of the cycle, each process has computed the correct state trajectory up to the global virtual time.

7.2. Detecting the End of a Parallel Segment

Algorithm 7.1 will compute many more speculative events than are strictly necessary. Consider the event sequences shown in Figure 7.2. There are two processes called A and B which both find outputs after some number of state transitions. Process A finds an output event very quickly and that output has the smaller time stamp. Hence, the global virtual time calculation will return the time stamp of process A's output. However, process B does not find an output until some time in the future, and then only after processing several internal events. Unfortunately, all of the work done by B after process A has found its output is wasted.

[Steinman 1992] describes a solution to this problem which uses a notification message that is sent by each processor when it finds an output. This basic scheme approximates the next global output event time via an incremental minimum calculation whose inputs are the output event times of each processor. The processes contribute their

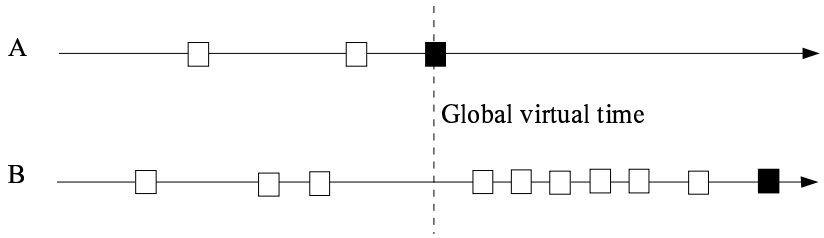

Figure 7.2. Process A waits for process B to find its next (incorrect) output. The black squares are output events and the white squares represent internal events without associated outputs.

output event time as it is found, with the incremental minimum operation terminating when all processes have either found an output or their time of next event exceeds the minimum value computed thus far.

If the parallel computer supports a lightweight broadcast mechanism, an incremental minimum can be computed in a straightforward way. Each process maintains an approximation of the next global output time. Initially, this value is set to infinity at the beginning of each cycle in Algorithm 7.1. Following the computation of an internal event, the process checks to see if a notification message has been received. If a notification message has been received, then the process computes a new minimum approximation that is the smaller of its current approximation and the time stamp included in the notification message.

The process stops computing internal events if it has found an output, or its time of next event is larger than its minimum approximation. If the process has found an output and its output event time stamp is less than the approximated minimum, then the process broadcasts a notification message containing its output event time. Otherwise, the process sends no notification message.

It may be the case that, when the next cycle begins, there are notification messages from the previous cycle still resident in the communication system (either in transit or stored in an operating system kernel buffer). In order to prevent premature termination of the output search, a cycle counter is included with each notification message. The cycle counter at each processor is incremented at the beginning of each cycle. Notification messages, whose cycle counter has a value less than the cycle counter at the receiving process, are simply discarded.

The procedure described above is listed as Algorithm 7.2. We assume that there is a broadcast function that delivers a message to all processes. On receipt of the message, the broadcasted message is stored in a buffer until explicitly requested. Such a broadcast mechanism can be constructed on a network of workstations using a multicast address and the Berkely socket library send() and recvfrom() function calls [Gray 1997]. It is not required that the broadcast mechanism be reliable or FIFO.

Algorithm 7.2

```
1.  Initially
2.      c := 0
3.      t _min := ∞

4.  when start_cycle()
5.      t_min := ∞
6.      c := c+1
7.  end

8.  when signal(t)
9.      if (t < t_min)
10.             t_min := t
11.             broadcast((t,c))
12.     end
13.end

14.when get_min()
15.     test for available broadcast message (t,d)
16.     if (message is available)
17.             remove (t,d) from the receive buffer
18.             if (t < t_min and c ≥ d)
19.                     t_min := t
20.             end
21.     end
22.     return t_min
23.end
```

It is a simple matter to augment Algorithm 7.1 to take advantage of the notification procedure described by Algorithm 7.2. The resulting risk-free optimistic simulation algorithm is listed as Algorithm 7.3.

Algorithm 7.3 (Breathing Time Buckets simulation)

```
1.  s := Φ
2.  S_k := {s}
3.  while (terminating condition not met)
4.        start_cycle()
5.        while(out(s).v = Φ and eN(s).t < get_min())
6.              s := push(s,eN(s))
7.              S_k := S_k ∪ {s}
8.        end
9.        if (out(s).v ≠ Φ)
10.             signal(out(s).t)
11.       end
12.       GVT := min(eN(s).t)
13.       if (GVT = eN(s).t)
14.             y := out(s).v
15.       else
16.             y := Φ
17.       end
18.       x := exchange_io(y)
19.       if (x ≠ Φ)
20.             S_k := S_k - { s | s.t ≥ x.t and s ∈ S_k }
21.             s := max S_k
22.             if (GVT = eN(s).t)
23.                   s := push(s,(x,GVT)•eN(s))
24.             else
25.                   s := push(s,(x,GVT))
26.             end
27.             S_k := S_k ∪ {s}
28.       end
29.end
```

8. DEVS APPROXIMATIONS OF CONTINUOUS SYSTEMS

In this chapter, we will consider the use of quantized state systems to simulate stationary, continuous systems that are described by ordinary differential equations. For this purpose, two quantized state systems are derived from the Adams-Bashforth family of integration schemes (see, e.g., [Ralston 2001]). The first scheme is identical to the QSS method presented in [Kofman 2003]. The second scheme exhibits errors that are proportional to the square of the integration quantum.

The use of the quantized state integrators for simulating systems of equations will be discussed, with an emphasis on solving partial differential equations in which approximation of the spatial derivatives gives rise to a system of ordinary differential equations. Using the QSS method to solve systems of equations is considered in depth in [Kofman 2003]. Some additional properties of the QSS method when applied to linear systems are considered here.

The presentation of the QSS method (called DAB1 here to emphasize the focus on interpolation) differs from that in [Kofman 2003] in order to explore, if only briefly, the use of interpolation to improve the order of quantized state integration schemes. In particular, the second order scheme constructed in this chapter approximates the derivative function using piece-wise linear, as opposed to piece-wise constant, functions. It is tempting to explore the use of Lagrangian interpolation to construct even higher order schemes, but such a study is well beyond the scope of this dissertation.

8.1. Concepts

Consider a stationary system described by the ordinary differential equation

8-1) $\dot{y} = f(y)$.

A simple numerical scheme for simulating this system is the explicit Euler formulae

8-2) $y_{n+1} = y_n + hf(y_n)$,

where h is the integration time step. The discrete time system described by (8-2) will produce an approximate solution to (8-1) if some particular assumptions concerning f(y) and h are met (see, e.g., [Ralston 2001]). Figure 8.1 shows the true and approximate behavior, computed using (8-2) with time step h = 1.0, for the stable stationary system

8-3) $\dot{y} = \sin(y)$.

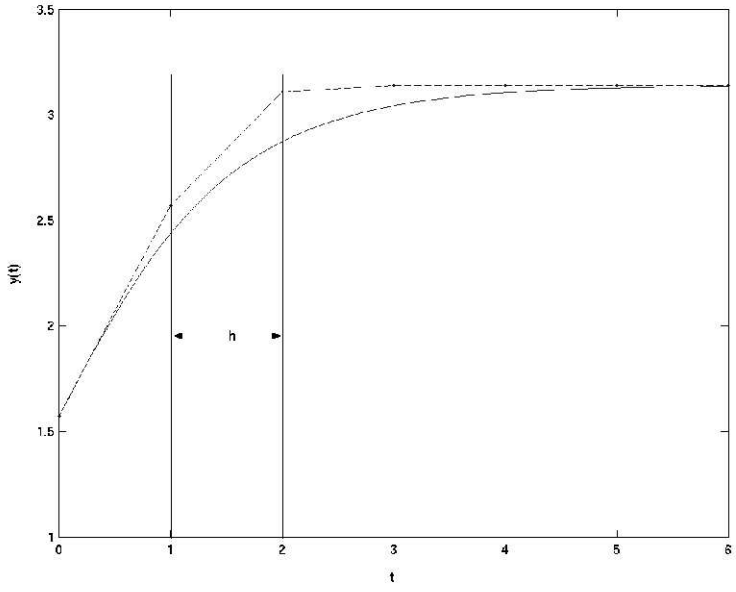

Figure 8.1. Solution to $\dot{y} = \sin(y)$ (solid line) and its approximate solution by explicit Euler (dotted line).

Suppose that instead of generating approximate solutions at discrete points in time, we approximate the solution by looking at significant changes in the system state. The magnitude of change in the solution that we consider significant is called an integration quantum (or quantum). We can define the integration quantum by

8-4) $D = |y_n - y_{n+1}|,$

where D is the desired change in the solution at each step of the computation. The time required for such a change to occur can be approximated by rearranging (8-2) to solve for h. Doing so, we find that

8-5) $h = D / |f(y)|.$

If $f(y) = 0$, we let $h = \infty$. Since we have assumed that y(t) is a stationary system, $f(y) = 0$ indicates that an equilibrium has been reached and, consequently, the solution will not change. So no information is lost by setting $h = \infty$.

We can construct a quantized state integration scheme by substituting (8-5) into (8-2) and keeping track of the sign of the derivative to ensure that the solution moves in the proper direction. This gives us the system

8-6) $y_{n+1} = y_n + D \operatorname{sgn}(f(y_n)),$

which approximates successive values of the continuous system. The formula

8-7) $t_{n+1} = t_n + D / |f(y_n)|$

gives the time at which approximate states occur. The result of using (8-6) and (8-7) to generate an approximate solution of (8-3) with $D = 1$ is shown in Figure 8.2.

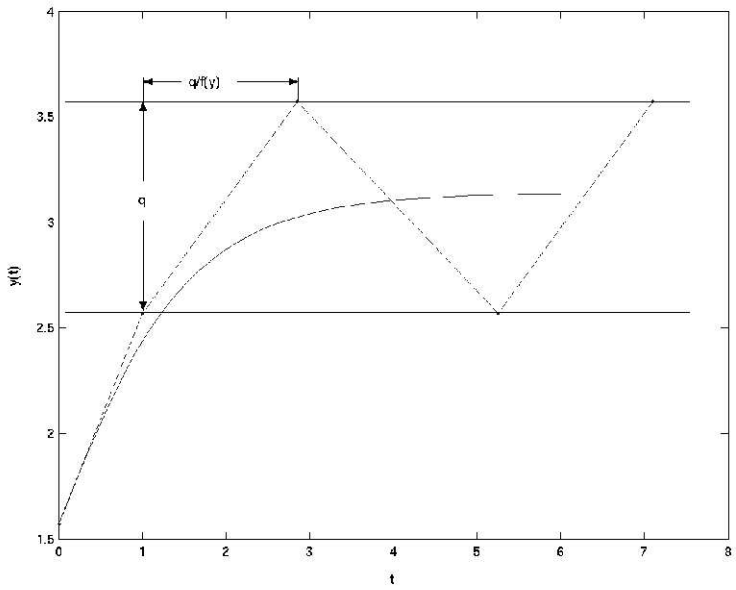

Figure 8.2. Solution to $\dot{y} = \sin(y)$ (solid line) and its approximate solution by the quantized integration scheme (8-6) and (8-7).

Formulae (8-6) and (8-7) illustrate the approach that will be used to develop quantum based integration schemes later in this chapter. We will start with an explicit, fixed step discrete time scheme and derive a time advance function by solving the scheme for h in terms of the integration quantum. The derivative approximation used by the discrete time scheme will be employed to determine the sign of the derivative in the quantized state scheme. Bounds on the global and local truncation error for the quantized state schemes can be developed in much the same fashion as the discrete time scheme. However, the proof of convergence will differ significantly due to the non-linear nature of (8-6).

8.2. Adams-Bashforth Type Quantized State Systems

We will be interested in solving an ordinary differential equation in the form of (8-1). The discrete time Adams-Bashforth methods provide one avenue for solving this type problem. They are derived by approximating the differential function f(y) using Newton's backward interpolation scheme, in which we know the value of f(y) at a point in time t and, possibly, at some number of points in the past (for details, see [Ralston 2001]). Let h be the difference in time between points at which the value of $f(y_j)$ and $f(y_{j-1})$ are known. Then the Adams-Bashforth method of order p (ABp) is

$$8\text{-}8) \qquad y(t+h) = y(t) + h\sum_{j=1}^{p} \alpha_{p,j} f(y(t+(1-j)h)),$$

where the $\alpha_{p,j}$ are selected so that the integral approximation is exact when f(y) is a polynomial of degree p or less.

When p = 1 or 2, a DEVS-Adams-Bashforth method of order p (DABp) can be constructed from (8-8) as follows. Let the quantum size D = |y(t+h)-y(t)|. The time advance of DABp immediately following an internal event can be obtained by rearranging (8-8) to solve for h. The state transition function for the DABp model is given by (8-8) with the time step being replaced by elapsed time.

A DABp can be expressed as a DEVS atomic model in the following way. The model state is described by the tuple (q,q_l,f_0,σ) for DAB1 and $(q,q_l,f_0,f_{-1},\sigma)$ for DAB2. The current value of the solution is q, q_l is the solution at the last quantum crossing, the f_i are the previously computed values of the derivative function, and σ is the time until the next

internal event occurs. The model accepts a single input x which is the value of the derivative function f(y), where external events coincide with changes in the derivative value.

The AB1 and AB2 schemes, respectively, are

8-9) $y_{n+1} = y_n + hf_n$, and

8-10) $y_{n+1} = y_n + (h/2)(3f_n - f_{n-1})$.

Solving for h in terms of quantum D yields the functions

8-11) $\tau_1(D, f_0) = \frac{D}{|f_0|}$, and

8-12) $\tau_2(D, f_0, f_{-1}) = \frac{2D}{|3f_0 - f_{-1}|}$.

The state transition function is computed using the last state q, an elapsed time e, and the known values of the derivative. The state transition functions are

8-13) $\Delta_1(q, e, f_0) = q + e f_0$, and

8-14) $\Delta_2(q, e, f_0, f_{-1}) = q + \frac{e}{2}(3f_0 - f_{-1})$.

The DAB1 scheme is described by

$\delta_{int}((q,q_l,f_0,\sigma)) = (\Delta_1(q,\sigma,f_0),\Delta_1(q,\sigma,f_0),f_0,\tau_1(D,f_0,))$,

$\delta_{ext}((q,q_l,f_0,\sigma),e,x) = (\Delta_1(q,e,f_0),q_l,x,\tau_1(D-|\Delta_1(q,e,f_0)-q_l|,x))$,

$\delta_{con}((q,q_l,f_0,\sigma),x) = (\Delta_1(q,\sigma,f_0),\Delta_1(q,\sigma,f_0),x,\tau_1(D,x))$,

$\lambda((q,q_l,f_0,\sigma)) = \Delta_1(q,\sigma,f_0)$, and

$ta((q,q_l,f_0,\sigma)) = \sigma$.

Similarly, DAB2 is defined by

$$\delta_{int}((q,q_l,f_0,f_{-1},\sigma)) = (\Delta_2(q,\sigma,f_0,f_{-1}),\Delta_2(q,\sigma,f_0,f_{-1}),f_0,f_{-1},\tau_2(D,f_0,f_{-1})),$$

$$\delta_{ext}((q,q_l,f_0,f_{-1},\sigma),e,x) = (\Delta_2(q,e,f_0,f_{-1}),q_l,x,f_0,\tau_2(D-|\Delta_2(q,e,f_0,f_{-1})-q_l|,x,f_0)),$$

$$\delta_{con}((q,q_l,f_0,f_{-1},\sigma),x) = (\Delta_2(q,\sigma,f_0,f_{-1}),\Delta_2(q,\sigma,f_0,f_{-1}),x,f_0,\tau_2(D,x,f_0)),$$

$$\lambda((q,q_l,f_0,f_{-1},\sigma)) = \Delta_2(q,\sigma,f_0,f_{-1}), \text{ and}$$

$$ta((q,q_l,f_0,f_{-1},\sigma)) = \sigma.$$

For a single equation, the DAB1 method is exactly the difference equation described by (8-6) and (8-7). To see this, first note that we can discard the confluent and external transition functions since the single equation can receive no inputs. The internal transition function reduces to (8-6) since the elapsed time e in (8-13) is always equal to the time advance given by (8-11). Substituting (8-11) for e in (8-13) results in (8-6). The DAB2 method applied to a single differential equation can be described in a similar way by replacing Δ_1 by Δ_2 in (8-6) and τ_1 by τ_2 in (8-7).

When the DABp methods are expressed as atomic DEVS models, we can apply them to the solution of systems of equations. A system of equations can be represented as a network of DABp integrators and instantaneous functions (see [Kofman 2003] for treatment of networks containing DAB1 integrators).

8.3. Local Truncation Error of the DEVS-Adams-Bashforth Methods

The local truncation error for the DABp methods can be derived by methods similar to those used in classical error analysis. To illustrate the procedure, we use the method described in [Ralston 2001]. For multi-step integration schemes, of which the ABp methods are an example, the truncation error at the n^{th} iteration can be expressed as

$$8\text{-}15) \quad T_n = \int_{t_{n-p}}^{t_{n+1}} y^{[p+1]}(s) G(s) ds ,$$

where G(s) is an appropriate Peano kernel, and $y^{[p+1]}$ is the (p+1)-th time derivative of y(t) (see [Ralson 2001] for details). If G(s) does not change sign in the interval $[t_{n-p}, t_{n+1}]$, then we have

$$8\text{-}16) \quad T_n = \frac{y^{[p+1]}(\gamma)}{p!} \int_{t_{n-p}}^{t_{n+1}} G(s) ds ,$$

where $t_{n-p} < \gamma < t_{n+1}$. If G(s) does change sign in the interval $[t_{n-p}, t_{n+1}]$, then T_n can still be bounded as

$$8\text{-}17) \quad |T_n| \leq \frac{|y^{[p+1]}(\gamma)|}{p!} \int_{t_{n-p}}^{t_{n+1}} |G(s)| ds .$$

For the DAB1 method, the integral limits are taken to be t_n, the simulation time at the n^{th} simulation cycle, and $t_n + t_a$, the simulation time at the next simulation cycle. Here, t_a is the time advance and so is given by $\frac{D}{|\dot{y}(t_n)|}$. The Peano kernel for the AB1 method

is $G(s) = (t_n + t_a - s)$ when $s \in [t_n, t_n+t_a]$. Clearly, G(s) does not change sign in this interval. Plugging this into (8-16) and integrating gives

$$8\text{-}18) \quad T_n = \frac{-\ddot{y}(\gamma)\, t_a^{\,2}}{2}.$$

Substituting $t_a = \frac{D}{|\dot{y}(t_n)|}$ into (8-18), we find that the local truncation error is

$$8\text{-}19) \quad T_n = \frac{D^2\, \ddot{y}(\gamma)}{2|\dot{y}(t_n)|^2}.$$

The truncation error for the DAB2 method can be computed in the same fashion. In this case, the Peano kernel is

$$8\text{-}20) \quad G(s) = G_1(s) + G_2(s),$$

where

$$8\text{-}21) \quad G_1(s) = ((t_{n+1}\text{-}s)^2 - (t_n\text{-}s)^2 - 3t_a(t_n\text{-}s)) / 2 \qquad \text{for } t_{n-1} \le s < t_n$$

$$0 \qquad \text{otherwise,}$$

and

$$8\text{-}22) \quad G_2(s) = (t_{n+1}\text{-}s)^2 / 2 \qquad \text{for } t_n \le s \le t_{n+1}$$

$$0 \qquad \text{otherwise.}$$

Define the values $\alpha = |3\dot{y}_n - \dot{y}_{n-1}|$ and $\beta = |3\dot{y}_{n-1} - \dot{y}_{n-2}|$. If the time advance is non-increasing in the interval $[t_{n-1}, t_{n+1}]$ (i.e., the time advance at integration step n is at least as large as that at step n-1), then (8-20) does not change sign in that interval. Substituting (8-20) into (8-16), integrating, and then performing a series of algebraic simplifications ultimately yields the truncation error of the DAB2 method, which is

$$8\text{-}23) \quad T_n = \frac{D^3 y^{(3)}(\gamma)(6\alpha\beta + 2\beta^2 - 3\alpha^2)}{3\alpha^3\beta^2},$$

where $t_{n-1} < \gamma < t_{n+1}$.

8.4. Convergence of the DEVS-Adams-Bashforth Methods

Consider a single equation of the form (8-1). Suppose we use a DABp integrator to approximate a solution on an interval of length I and that the the real and computed sign of f(y) agree and do not change in the interval. Let Y_n be the true solution at the n^{th} iteration and y_n the computed solution. A computation will result in a sequence of approximations to the real solution given by

$y_0 = Y_0$

$y_1 = y_0 + D\ \mathrm{sgn}(f(y_0)) = Y_0 + D + T_1$

$y_2 = y_1 + D\ \mathrm{sgn}(f(y_1)) = Y_0 + 2D + T_1 + T_2$

.....

The sign function has the effect of suppressing numerical errors in the calculation of f(y) so long as a sign error does not occur. This differs significantly from the classical Adams-Bashforth methods in which numerical errors in the computation of f(y), multiplied by the time step, appear in the next computed state. Error propagation in the DABp methods occurs via the dependence of the truncation error on previously computed values of f(y(t)). For example, expanding T_2 in the above sequence gives

$$T_2 = \frac{D^2 \ddot{Y}(\gamma_2)}{2\left|f(y_1)\right|^2} = \frac{D^2 \ddot{Y}(\gamma_2)}{2\left|f(Y_0 + D + T_1)\right|^2} = \frac{D^2 \ddot{Y}(\gamma_2)}{2\left|f(Y_0 + D + \frac{D^2 \ddot{Y}(\gamma_1)}{2\left|f(y_0)\right|^2})\right|} .$$

The value of Y(t) after n computational steps is described by

$$8\text{-}24) \quad Y_n = Y_0 + \sum_{i=1}^{n} T_i + nD ,$$

where the T_i are dependent on the response of the derivative function to the global error in the previous step. If we assume that the derivative function is defined everywhere, then T_i will always take on some finite value, and we can assume some upper bound on the magnitude of that value in the course of the computation.

The number of steps required to compute through an interval of length I can be bounded as

$$8\text{-}25) \quad n \leqslant \frac{I}{\min \tau_p} ,$$

where min τ_p is the smallest value produced by (8-11) or (712) in the course of the computation. The error accumulated at the end of the interval can then be bounded as

$$8\text{-}26) \quad \epsilon_n \leqslant \frac{I \max T_i}{\min \tau_p} ,$$

where max T_i is the largest local truncation error observed during the computation. Using (8-26), we bound the final error of the DAB1 approximation as

$$8\text{-}27) \quad \epsilon_n \leqslant \frac{I D \ddot{Y}_{max} \left|\dot{y}_{max}\right|}{2\left|\dot{y}_{min}\right|} ,$$

where $\dot{y}_{max}$ and $\dot{y}_{min}$ are the largest and smallest, respectively, computed values of the derivative function. Similarly, the final error for a DAB2 approximation is bounded as

$$8\text{-}28) \quad \epsilon_n \leq \frac{1}{2} I D^2 Y^{(3)}_{max} \gamma_{max},$$

where

$$8\text{-}29) \quad \gamma = \frac{6\alpha\beta + 2\beta^2 - 3\alpha^2}{3\alpha^3\beta^2}$$

and the α and β values are functions of the computed y_i. This latter bound assumes, of course, that the time advance function is non-increasing in the interval I.

Figures 8.3a and 8.3b show the absolute error as a function of the quantum size for the DAB1 and DAB2 solutions to $\dot{y} = -y$ with y(0) = 1. These results agree nicely with the error behavior predicted by (8-27) and (8-28).

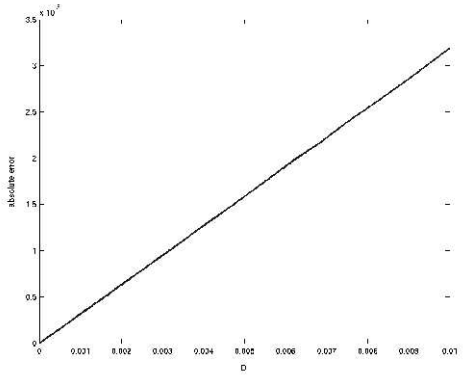

Figure 8.3a. The absolute error in the DAB1 approximation to $\dot{y} = -y$ with y(0) = 1 as a function of the integration quantum.

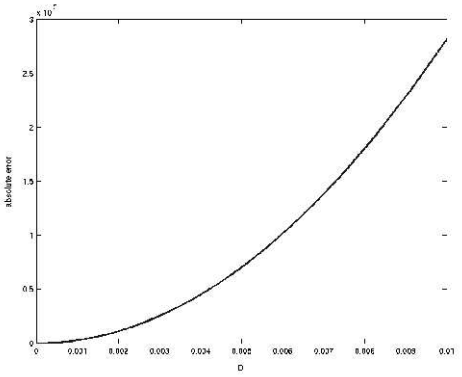

Figure 8.3b. The absolute error in the DAB2 approximation to $\dot{y} = -y$ with y(0) = 1 as a function of the integration quantum.

When the truncation error is known, formulae (8-26) gives us a way of computing the final error when computing over an interval where the sign of the derivative function is constant and can be computed without error. A bound on the final error when sign changes are allowed can be derived if we are able to make a handful of assumptions about y(t). Let us suppose that the following statements are true.

1) There exists a Lyapunov function V(y(t)), and

2) V(y(t)) is strictly decreasing in t.

From these assumptions it immediately follows that, for a given initial state y_0, there exists a unique final state $\bar{y}$ such that $f(\bar{y}) = 0$. Moreover, there exists an $\varepsilon > 0$ such that $|\bar{y} - y_0| < \varepsilon$ implies that $y(t) \rightarrow \bar{y}$ as $t \rightarrow \infty$. The following fact will also prove useful.

Lemma 8.1 If V(y(t)) is strictly decreasing in time and $|y(t) - \bar{y}| < \varepsilon$, then $|y(t) - \bar{y}|$ is strictly decreasing and f(y(t)) is in the direction of $\bar{y}$.

Proof Let $|y(t_1) - \bar{y}| = d_1$. For some $t_2 > t_1$, let $|y(t_2) - \bar{y}| = d_2 > d_1$. Since V(y(t)) is strictly decreasing in time, it must be that $V(y(t_2)) < V(y(t_1))$. Now, for some time $t_3 > t_2$, it must be the case that $|y(t_3) - \bar{y}| = d_1$ since y(t) is approaching $\bar{y}$ as t becomes large. At this point, we have $V(y(t_3)) = V(y(t_1))$. This contradicts the strictly decreasing property of V since $V(y(t_3)) > V(y(t_2))$ but $t_3 > t_2$. So $|y(t) - \bar{y}|$ is strictly decreasing in time. It follows immediately that f(y) is in the direction of $\bar{y}$. ■

Starting from an initial state y_0, suppose we compute through n steps, at which point the sign of the derivative (either in the real or computed solution) is changed. For a

continuous stationary system, a change in the sign of the real derivative implies that y(t) = $\bar{y}$, and so f(y(t)) = 0. Using (8-26), we can bring the computed solution as close as desired to equilibrium before a change occurs in the sign of the computed derivative. That is, the computed solution can be brought arbitrarily close to $\bar{y}$. Let the computed solution at the n^{th} step be $y_n = \bar{y} + \Delta_n$ and choose $|\Delta_n| < \varepsilon$.

Consider the DAB1 integrator. From Lemma 8.1, it is known that the derivative will change sign in such a way as to move the solution towards equilibrium. That is, $f(y_n)$ will have a sign opposite that of the error. If we assume that $\Delta_n > 0$ and choose $D < \varepsilon$, then the next computed state will have a value $y_n - D = \bar{y} + \Delta_n - D$. Since $|\Delta_n - D| < \varepsilon$, we have that $|y_{n+1} - \bar{y}| < \varepsilon$. So the computed solution will oscillate about the equilibrium, with the magnitude of the deviation from equilibrium being bounded by $\max(D,|\Delta_n|)$. Since the real solution is approaching equilibrium, this gives us an upper bound on the final error.

A more complex argument must be made for the bounded error of the DAB2 integrator. As before, at the end of any interval with a derivative of constant sign, we can select a final computed state that is arbitrarily close to equilibrium. Without loss of generality, assume that the sign of the computed derivative was negative prior to the sign change. When the sign change occurs, the new sign is given by $3f(y_n) - f(y_{n-1}) = 3f(\bar{y} + \Delta_n) - f((\bar{y} + \Delta_n + D)$. When the sign change occurs, it will be the case that either $3f(\bar{y} + \Delta_n) = 0$ (i.e, that $\Delta_n = 0$) or $3f(\bar{y} + \Delta_n)$ and $f(\bar{y} + \Delta_n + D)$ will have opposite signs.

First, consider the case where $\Delta_n = 0$. If we select $D < \varepsilon$, the sign of $f(\bar{y} + D)$ will be towards the equilibrium (Lemma 8.1), and so $-f(\bar{y} + D)$ is pointing away from the equilibrium. At the next computational step, we have $y_{n+1} = \bar{y} + D$. Now, $3f(y_{n+1}) - f(y_n) = 3f(\bar{y} + D) - f(\bar{y}) = 3f(\bar{y} + D)$ which is in the direction of $\bar{y}$ since $D < \varepsilon$. So $y_{n+2} = \bar{y} + D - D = \bar{y}$. It can be seen that the computed solution will oscillate about $\bar{y}$ with the magnitude of the oscillation bounded by D. Since the real solution is approaching equilibrium the global error can be bounded by D.

If $\Delta_n \neq 0$, then $3f(\bar{y} + \Delta_n)$ and $-f(\bar{y} + \Delta_n + D)$ will have equal signs and so $3f(\bar{y} + \Delta_n) - f(\bar{y} + \Delta_n + D)$ points towards the equilibrium and has an opposite sign to Δ_n. The state at the next computational step can be computed as $\bar{y} + \Delta_n - D$. Choosing $D < \varepsilon$ gives $|\Delta_n - D| < \varepsilon$. Hence the computed solution will oscillate near the equilibrium, with the magnitude of the deviation from equilibrium being bounded by $\max(D, |\Delta_n|)$. Since the real solution is approaching equilibrium, this gives us an upper bound on the final error.

8.5. Linear Stability for DAB1 and DAB2

Consider the linear system $\dot{y} = -ky$. For $k > 0$, this system is globally asymptotically stable. That is, for any initial state, $y(t) \rightarrow \bar{y}$ as $t \rightarrow \infty$. From the preceding arguments, it can be seen that any integration quantum will cause the system to

oscillate about $\bar{y}$, with the magnitude of the oscillations going to zero as D goes to zero. Figures 8.4 and 8.5 illustrate this phenomena for the DAB1 and DAB2 methods.

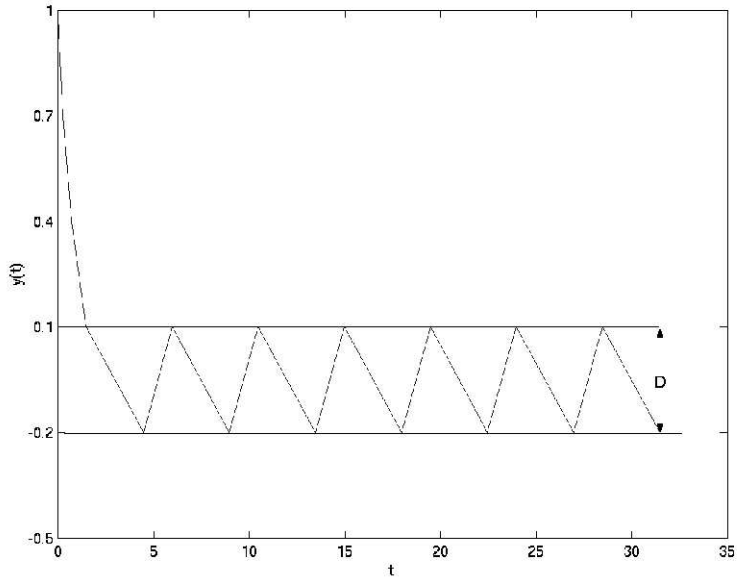

Figure 8.4. Oscillations in the solution of dy/dt = -y with y(0) = 1 and D = 0.3 using DAB1.

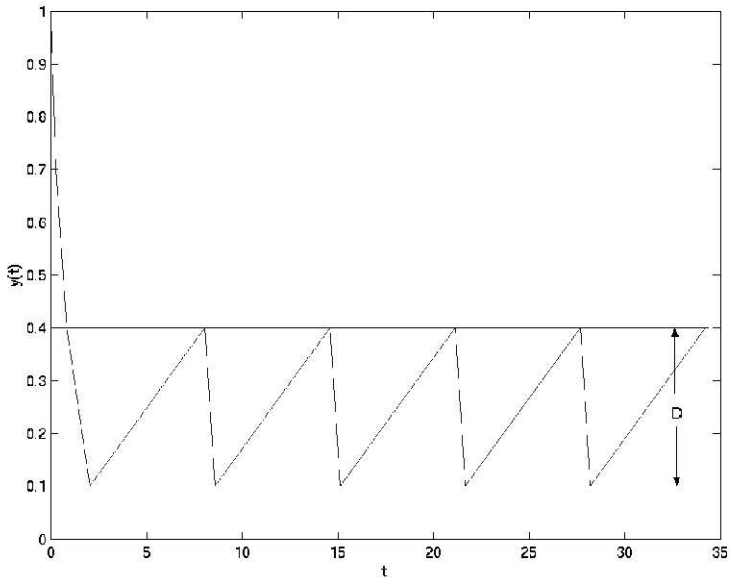

Figure 8.5. Oscillations in the solution of dy/dt = -y with y(0) = 1 and D = 0.3 using DAB2.

8.6. Vector Valued States

When y(t) is a vector, (8-1) describes a system with several state variables. This problem could be tackled by integrating each state variable using a scalar integrator, with the interdependencies of the state variables being represented by a coupled DEVS model. Another approach is to apply a DABp method with a modified time advance function. The new time advance function will use a quantum volume to determine when the next significant event will occur. When the state of the system leaves one volumetric unit in the state space and enters an adjacent unit, an internal event will occur in the DEVS model.

Suppose that y has n components $y_1, y_2, \ldots, y_n$. The state space of the DABp becomes a tuple of vectors

$q = (q_1\ q_2 \ldots q_n)$, where q_k is the value of the k^{th} state variable,

$q_l = (q_{l-1}, \ldots, q_{l-n})$, where q_{l-k} is the value of the k^{th} state variable at the time of the last quantum threshold crossing,

$f_i = (f_{i-1}, \ldots, f_{i-n})$, where f_{i-k} is the i^{th} known value of the derivative for the k^{th} state variable, and

$\sigma = (\sigma_1, \ldots, \sigma_n)$, where σ_k is the time until the k^{th} state variable crosses the quantum volume boundary, while traveling along the k^{th} axis.

The quantum also becomes vector valued, where $D = (D_1, \ldots, D_n)$ and D_k is the integration quantum for the k^{th} state variable.

Suppose we wish to describe quantum volumes by rectangles, cubes, or some higher-dimensional analog of such an object. Let τ_{p-k} be (8-11) or (8-12) applied to the k^{th} component of the state vector. Then the vector valued analog of (8-11) and (8-12) is

$$8\text{-}30)\ \tau_p(D,f_0,f_{-1},\ldots,f_{-(p-1)}) = \min \{ \tau_{p-k}(D_k,f_{0-k},f_{\{-1\}-k}, \ldots, f_{\{-(p-1)\}-k}) : k\in[1,n] \}.$$

Figure 8.6 illustrates (8-30) when there are two state variables.

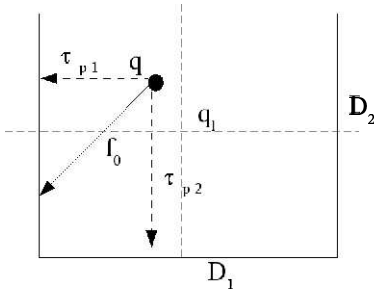

Figure 8.6. Movement of a solution within a quantum volume for a two-dimensional system.

8.7. Partial Differential Equations

Many numerical solutions to partial differential equations with time and space components have two distinct and independent aspects. The first aspect of a numerical method is the approximation of spatial derivatives. After applying a spatial approximation technique, we are left with a system of coupled ordinary differential equations. The second aspect of the numerical method is then to select a technique for integrating through time.

A set of coupled differential equations can be described as a multi-component system, with each component describing one differential equation or algebraic coupling equation. The spatial discretization of a partial differential equation can generate such a system. If we replace integration terms by their discrete event approximations (i.e., a DABp integrator), then we immediately arrive at a multi-component DEVS model that approximates the partial differential equation. This can be simulated using an appropriate DEVS simulator (parallel or otherwise) to generate numerical solutions.

The use of DEVS as a solution technique for partial differential equations is attractive due to the innate ability of discrete event integrators to adjust their level of computational effort to match the rate at which the solution is changing. This results in a very efficient 'front' or 'activity' tracking technique. This technique is widely applicable since it relies on intrinsic properties of the time integration scheme, rather than on some aspect of the spatial discretization. The activity tracking capability of DAB1 to finite difference approximations of the heat equation is considered in [Jammalamadaka 2003].

8.8. Application of DAB1 to Finite Difference Schemes

The use of finite differences to approximate spatial derivatives in a partial differential equation gives rise to an input free continuous linear system that can be expressed as

$$\dot{x}(t) = A\,x(t), \tag{8-31}$$

where the state vector has one element for each spatial grid point. For example, the heat equation in one dimension with coefficient of diffusion k is given by

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}, \tag{8-32}$$

where u(x,t) is the temperature at time t and position x. The spatial derivative can be approximated using center differences, this giving

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2}, \tag{8-33}$$

where Δx is the spatial resolution of the approximation. Substituting (8-33) into (8-32) gives a system of coupled ordinary differential equations

$$8\text{-}34) \quad \frac{\partial u(x_i,t)}{\partial t} \approx k \frac{u(x_{i+1},t) - 2u(x_i,t) + u(x_{i-1},t)}{(\Delta x)^2},$$

where x_i is the position $i(\Delta x)$.

For the heat equation, it is necessary to specify boundary conditions so that the problem is well posed. When the boundary conditions are fixed (i.e., the edge cells are assumed to have a constant temperature), then (8-34) is a stable linear system.

Another choice of boundary conditions that result in a marginally stable linear system is

$$8\text{-}35) \quad u(0,t) = u(\Delta x,t) \text{ and } u(N\Delta x,t) = u((N-1)\Delta x,t),$$

where the cells 1, 2, ... , N-1 are in the interior of the computational region, and the cells 0 and N are at the boundaries. Since the system is only marginally stable, the cumulative effects of perturbations from the equilibrium condition is to cause the computed solution to drift away from the true solution when the computed solution is near equilibrium.

This phenomenon is avoided in first order discrete time approximation of (8-34) by adhering to the Courant condition. The unconditional linear stability of DAB1 ensures a stable computational scheme for a stable form of (8-34). For marginally stable systems, sufficient conditions for stability are presented in what follows, although they turn out to be much less useful than the Courant condition.

Given a matrix A, the element in row i and column j is denoted by $(A)_{ij}$. Similarly, for a vector x, the i^{th} element of x is denoted by $(x)_i$. Given a vector x, denote a vector norm of x by $|x|$. The p-norm of x is denoted by $|x|_p$. Let

$$\dot{x}(t) = \alpha A x(t) \qquad 8\text{-}36)$$

be a linear system in which $(A)_{ij}$ is an integer, α is a real number, and $\bar{x}$ an equilibrium of (8-36). Let $x_q(t)$ be the output of the network of DAB1 integrators used to approximate (8-36) using integration quantum D. Moreover, select D such that $Ax_q(0) = Dz$, where z is a vector whose elements are integers. Then the following statements are true:

Lemma 8.2 For all $t > 0$, $Ax_q(t) = Dz'$ where z' is a vector whose elements are integers.

Proof The lemma follows immediately from the definition of the DAB1 output function and the integer coefficients of A. ■

Lemma 8.3 $|Ax_q(t)|_1$ and $|Ax_q(t)|_\infty$ are integer valued.

Proof The lemma follows from Lemma 8.2 and definition of the 1- and ∞-norms. ■

Theorem 8.1 Suppose $|Ax_q(t)|$ is integer valued and $x_q(t)$ is such that $|Ax_q(t)| = 0$ or there exists a positive value τ such that, for all $t' \leq t+\tau$, $|Ax_q(t')| < |Ax_q(t)|$. Then for some $t \geq 0$, $x_q(t)$ is an equilibrium of (8-36).

Proof Suppose $|Ax_q(t)|. = 0$. Then it must be the case that $Ax_q(t) = 0$ and so $x_q(t)$ is an equilibrium of (8-36). Now, assume that the theorem holds when $|Ax_q(t)| = n$ and

consider the case in which $|Ax_q(t)| = n+1$. For some $\tau > 0$, $|Ax_q(t+\tau)| \geq n$. Applying the induction hypothesis, it can be seen that there exists a time $t' > t+\tau > t > 0$ such that $Ax_q(t') = 0$. That is, $x_q(t')$ is an equilibrium of (8-36). ■

Corollary 8.1 If (8-36) is stable and the hypothesis of Theorem 8.1 holds, then steady state solution of (8-36) is the steady state solution of its DAB1 approximation.

Proof The corollary follows from Theorem 8.1 and the uniqueness of $\bar{x}$ for a stable linear system. ■

Theorem 8.1 (and Corollary 8.1 in particular) state that if

1) the initial values of the derivatives are integer multiples of the integration quantum, possibly disregarding some real constant that is common to all of the initial derivatives, and

2) an integer valued norm of the the component derivatives is ultimately decreasing,

then the steady state solution of (8-36) is the steady state solution of its DAB1 approximation.

Example Consider an ordinary differential equation $\dot{x} = k\,x$, where $k < 0$ and $x(0) = 1$. Let $D = 1$. The first state transition occurs at $1/|k|$ and we can compute $x_q(1/k) = x(0)-D = 0$, which is the equilibrium of $x(t)$.

Now pick the integration quantum D such that $x_0 = zD$, where z is a positive integer. Using (8-6), $x_q(t)$ can be written as a difference equation whose free variable is the number of internal events. So we have $x_{n+1} = x_n - D$ with $x_0 = zD$, z a positive integer.

So $x_{n+1} = (z\text{-}n)D$ which is strictly decreasing and reaches steady state after z internal events. Consequently, $|kx_q(t)|$ is strictly decreasing or $|kx_q(t)| = 0$.

Even if $k = 0$ (i.e., the system is only marginally stable), (3-6) becomes $x_{n+1} = x_n = zD$ and so the quantized state approximation is marginally stable as well. In the marginally stable case, $|kx(t)| = 0$ and $|kx_q(t)| = 0$ as well.

Example Consider the stable linear system

$$\dot{x}(t) = \begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix} x(t) = Ax(t).$$

Consider $|Ax_q(t)|_1 = |x_2 - x_1| + |-x_2| \leq 2|x_2| + |x_1|$. To see that the hypothesis of Theorem 8.1 holds, first notice that $x_q(t) = 0$ is the equilibrium and $|A0|_1 = 0$. Case analysis can be used to show that $|Ax_q(t)|$ is decreasing when $x_q(t) \neq 0$.

Case 1 $x_1 > x_2$ and x_1 undergoes an internal event. Since $x_2 - x_1 < 0$, one quantum is subtracted from x_1, causing $|x_2 - x_1|$ to decrease by a single quantum. The case in which $x_1 < x_2$ can be addressed with a symmetric argument.

Case 2 x_2 undergoes an internal event. In this case, $|x_2|$ decreases by one quantum. Consequently, $2|x_2| + |x_1|$ decreases by two quantum. Repeated internal events must ultimately result in a decrease in $|Ax_q(t)|_1$. Note that if $2|x_2| + |x_1| = 1$, case 2 can not apply since $|x_2|$ must be zero and, consequently, is passive. This rules out oscillation about the steady state due to the persistence of case 2.

Case 3 x_1 and x_2 undergo a confluent event. In this case, $|x_2|$ decreases by one quantum. Since $|x_1|$ can change by at most one quantum, it must be that $2|x_2| + |x_1|$ decreases by one quantum ($|x_1|$ increases) or three quantum ($|x_1|$ decreases). So repeated

confluent events must ultimately result in a decrease in $|Ax_q(t)|_1$. Note that if $2|x_2| + |x_1| = 1$, Case 3 can not apply since $x_2 = 0$ and is passive. If $2|x_2| + |x_1| = 2$, then $|x_1| = 0$ and $|x_1|$ must increase by one quantum. Hence, the decrease in $2|x_2| + |x_1|$ is by one quantum. This rules out oscillation about the steady state due to the persistence of Case 3. It can be concluded that $|Ax_q(t)|_1$ is strictly decreasing as a function of the number of events, or that x_q is at an equilibrium. So the hypothesis of Theorem 8.1 is satisfied.

The utility of Theorem 8.1 is hampered in two ways. First, an appropriate integration quantum must be found. On a digital computer, this task can be reduced to finding a common divisor of a set of integers, where the integers are acquired by scaling the initial data (which is of finite precision). For example, given initial values 2.554 and 3.446, we can multiply by 1000 to get 2554 and 3446. The greatest common divisor of these numbers is 2. Diving by 1000, we arrive at a suitable integration quantum of 0.002. To check, we have 2.554/0.002 = 1277000 and 3.446/0.002 = 1723000. Notice that 0.2 would have worked as well.

In general, a set of integers will have 1 as a common divisor. Given a set of terminating decimals (i.e., a set of finite precision data points), a suitable quantum size can be had by using the inverse of the smallest power of ten multiple of the set. For example, given the numbers 1.03, 5.006, 1.0001, and 10.1, a suitable integration quantum is 0.0001. In this case, 1.03/0.0001 = 10300, 5.006/0.0001 = 50060, 1.0001/0.0001 = 10001, and 10.1/0.0001 = 101000. A larger quantum size, and so a faster computation, can be had by reducing the precision of the initial data.

A second task, not so easily completed as the first, is proving that some norm exists for the DAB1 approximation of a linear system such that the hypothesis of Theorem 8.1 holds. For a stable linear systems this is less critical since the unconditional linear stability of the DAB1 method ensures that the computed solution approaches, and stays close to, the true equilibrium of the solution. For marginally stable systems (e.g., the heat equation with adiabatic boundary conditions), Theorem 8.1 is the only means by which we can ensure that the computed solution will not drift away from an equilibrium point.

Illustrating Theorem 8.1 for large systems of equations via case analysis appears to be impractical. However, the sufficiency of the integer multiple condition for ensuring that the real equilibrium exists in the state space of the DAB1 approximation can be readily demonstrated.

Consider the heat equation in one dimension. The value of u(t,x) at each cell can be written in terms of its initial value α plus some multiple, q, of the quantum size. So at any particular cell, letting the subscript 0 indicate the cell of interest, and positive or negative subscripts indicate displacement to the right and left, we can write (8-34) as

$$8\text{-}37) \quad \frac{\partial u(x_0, t)}{\partial t} = k \frac{\alpha_{-1} + q_{-1} D - 2(\alpha_0 + q_0 D) + \alpha_{+1} + q_{+1} D}{(\Delta x)^2}.$$

In order for the discrete event system to reach the steady state of the continuous system, it is necessary for there to be integers values for q_{-1}, q_0, and q_{+1} such that (8-37) is zero. That is, there must exist integers of q_{-1}, q_0, and q_{+1} such that

8-38) $\alpha_{-1} + q_{-1} D - 2(\alpha_0 + q_0 D) + \alpha_{+1} + q_{+1} D = 0$.

Rearranging (8-38) gives

8-39) $(\alpha_{-1} - 2\alpha_0 + \alpha_{+1}) + (q_{-1} - 2q_0 + q_{+1}) D = 0$.

Rearranging the terms in (8-39), it can be seen that a sufficient condition for the state space of a DAB1 approximation of (8-34) to include the equilibrium of (8-34) is

8-40) $(\alpha_{-1} - 2\alpha_0 + \alpha_{+1}) = -(q_{-1} - 2q_0 + q_{+1}) D$.

Since the q_i are integers, this requires that $(\alpha_{-1} - 2\alpha_0 + \alpha_{+1})$ be an integer multiple of the integration quantum. For the adiabatic boundary conditions (8-35), the leftmost and rightmost cells must be such that

8-41) $(-\alpha_0 + \alpha_{+1}) = -(-q_0 + q_{+1}) D$, and

8-42) $(\alpha_{-1} - \alpha_0) = -(q_{-1} - q_0) D$

hold, respectively. Similarly, when the boundary conditions are such that u(0,t) = u(n,t) = 0, the left and right most cells must be such that

8-43) $(-2\alpha_0 + \alpha_{+1}) = -(-2q_0 + q_{+1}) D$, and

8-44) $(\alpha_{-1} - 2\alpha_0) = -(q_{-1} - 2q_0) D$

hold, respectively.

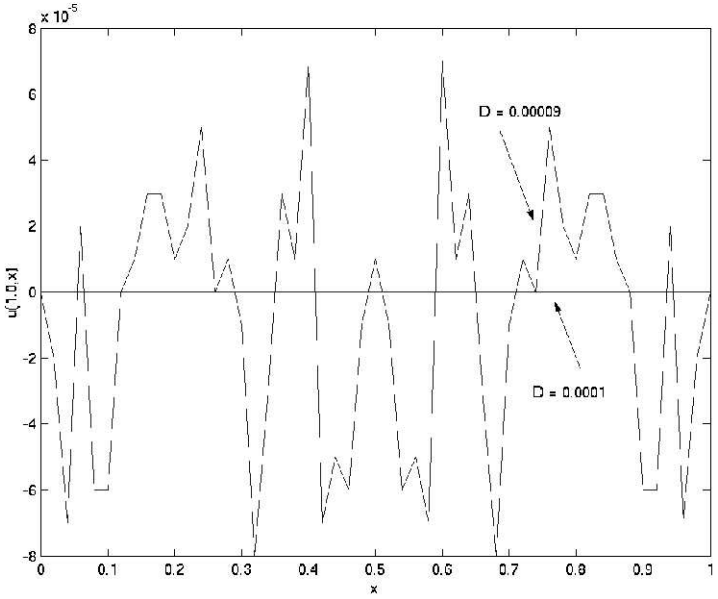

Figure 8.7. An illustration of the effect of the integration quantum on near steady state behavior for the heat equation in one dimension.

Figure 8.7 shows solutions to the heat equation computed using (8-34) with a DAB1 integrator. The initial data was computed using $u(0,x) = \sin(\pi x)$ with $x \in [0,1]$ and $\Delta x = 0.02$. The boundary conditions were taken to be adiabatic with the diffusion constant $k = 0.01$. The computation was run until $t = 1.0$. The initial data was taken with a precision of 10^{-4}. A quantum size of $D = 10^{-4}$ ensures that the true equilibrium of 0 is in the state space of the DAB1 approximation. The corresponding calculation stopped with each cell having a state whose magnitude was on the order of 10^{-17} using a double precision, IEEE floating point machine. A quantum size of $D = 9\times10^{-5}$ prevents the true equilibrium from existing in the DAB1 state space. The corresponding calculation oscillated about the equilibrium, but was unable to reach it.

If we are willing to discount confluent events that might occur in the course of solving the diffusion problem, then we can show that the computed solution will always go to an equilibrium of the real system given either adiabatic or fixed boundary conditions. As before, we proceed by case analysis, though restricting ourselves to internal events. Figure 8.8 illustrates the possible arrangements of a given cell and the relative values of its neighboring cells.

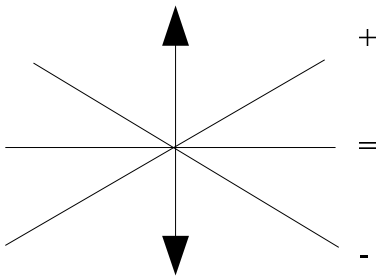

Figure 8.8. The 18 possible arrangements of a center cell and its neighboring cells.

For any particular cell, each immediate neighbor can have a value that is positive, negative, or zero with respect to that cell. The next state transition at the cell can cause its state variable to go up or down. Each case can be described by a three character string with the first character indicating the relative value of the left neighbor, the center character describing the direction of the state variable at the center cell, and the last character describing the relative value of the right neighbor. For example, the string (+++) corresponds to the upward V in Figure 8.8 and selection of the arrow pointing up. There are 18 such strings, corresponding to 18 different cases.

The following cases can not be realized by any computation: (+-=), (-+=), (=+-), (=-+), (+-+), (-+-), (=+=), and (=-=). There are two sets of symmetric cases with respect to the 1-norm. Consider these in turn.

Case 1 (++=), (--=), (=++), (=--), (++-), (-++), (+--), (--+) In this case, the difference between the cell and its left (right) neighbor is reduced by one quantum and the difference in the values of the cell and its right (left) neighbor is increased by one. Consequently, the 1-norm remains unchanged.

Case 2 (+++), (---) Here, the cell reduces the difference between its own value and its neighboring values by one, for a net decrease of two. Note that if this case holds, then the (local) value of the 1-norm must have been greater than or equal to two, thereby ruling out oscillations about the equilibrium due to the persistence of case 2.

It follows that the execution of the internal transition function at an interior cell can not cause the 1-norm of the derivative approximation to increase. To see that the 1-norm must eventually decrease or be equal to zero it is necessary to consider the edge cells. In the adiabatic case, any state transition at an edge cell will result in the 1-norm being reduced by one. Moreover, if the derivative value at the edge cells is zero, the system is at an equilibrium or case 2 must hold at some point in the interior. Similarly, for fixed boundary conditions the system is either at an equilibrium or case 2 must hold somewhere in the interior. So, assuming that a confluent transition does not occur during the computation, the hypothesis of Theorem 8.1 holds.

9. THE SOD SHOCK TUBE PROBLEM

The goal of this Chapter is to solve a conservation law with state vector **u** in one spatial dimension x whose form is

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial [f(\mathbf{u})]}{\partial x} = 0 . \qquad 9\text{-}1)$$

A solution can be approximated by dividing space into discrete cells and determining the flux of each state variable through a cell boundary. The motion of the solution is then computed by integrating over the net flux through a cell (for a brief overview see [Leveque 1996], [Sod 1978]).

This system can be modeled by a coupled DEVS that consists of three basic components. The first component is a DAB1 integrator that computes the integral at each cell using the modified time advance function described in Section 8.6. The second component computes the flux at a cell boundary using the outputs of the quantized integrators to the left and right of the boundary. The third is a summer that computes the net flux through a cell. The coupled model that realizes the discrete event approximation of the conservation law is shown in Figure 9.1.

The DAB1 integrator has already been described. The boundary flux model has two state variables $\mathbf{q}_{left}$ and $\mathbf{q}_{right}$, which are the estimates of the state variables in the left and right cells relative to the boundary. When the boundary flux model receives an input, it updates these values, computes the flux across the boundary, and then immediately generates an output that is the new flux value. The summers operate similarly by

remembering two values (the left and right fluxes) and responding to input by updating the two state variables and outputting the sum.

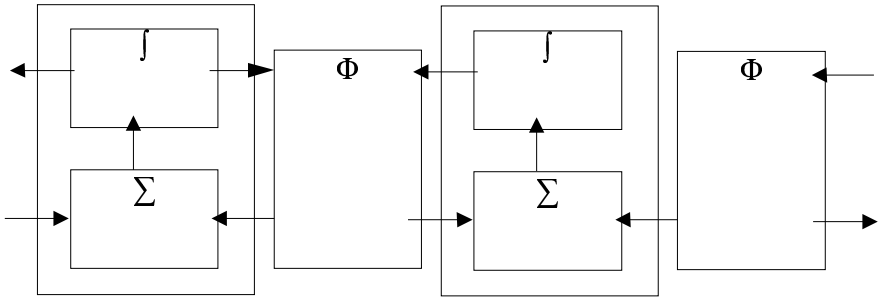

Figure 9.1. A DEVS coupled model that approximates the conservation law. The symbol ∫ denotes the quantized integrator, ∑ is the summer, and Φ is the flux model.

When the explicit Euler formulae is used to integrate the net flux, a necessary and sufficient condition for stability is that the wave velocity be less than half the length of a cell (see [Leveque 1996]). This condition arises from the need to update the flux function whenever the wave properties at a cell have propagated to either of the cell's neighbors.

In the discrete event scheme, the crux of the stability question seems to lie in the behavior of the flux approximation and, in particular, how quantum changes in a cell state correspond to changes in the net flux through a cell. When the flux approximation is linear, then it may be possible to apply Theorem 8.1 to find an integration quantum that ensures stability.

Along these lines, we can study a similar, though much simpler, problem and hope to gain some insight into the stability of proposed scheme.

Consider the advection problem

$$\frac{\partial u}{\partial t} = v \frac{\partial u}{\partial x} \qquad 9\text{-}2)$$

approximated by an up-winding finite difference scheme

$$9\text{-}3) \qquad \frac{\partial u}{\partial t} = v \frac{u_{m\Delta x} - u_{(m\pm 1)\Delta x}}{\Delta x}.$$

Similar to the heat equation, we can establish conditions under which the true equilibrium is in the state space of a DAB1 approximation of (9-3).

As before, the value of u(t,x) at each cell can be written in terms of its initial value plus some integer multiple of the integration quantum. So at any particular cell, letting the subscript 0 indicate the cell of interest and positive or negative subscripts indicate displacement to the right and left, we can write (9-3) as

$$9\text{-}4) \qquad \frac{\partial u(x_{0,} t)}{\partial t} = v \frac{\alpha_0 + q_0 D - (\alpha_{\pm 1} + q_{\pm 1} D)}{\Delta x},$$

where the q_i are integers. A sufficient condition for (9-4) to equal zero is

$$9\text{-}5) \qquad \alpha_0 + q_0 D - (\alpha_{\pm 1} + q_{\pm 1} D) = 0.$$

Rearranging (9-5) gives

$$9\text{-}6) \qquad \alpha_0 - \alpha_{\pm 1} = -(q_0 - q_{\pm 1}) D.$$

Assuming we can find an integration quantum to satisfy (9-6), it can be shown that the 1-norm applied to the DAB1 approximation of (9-3) satisfies the hypothesis of Theorem 8.1. As before, this is done by case analysis.

Case 1 Consider an interior cell u_k undergoing an internal event and using (without loss of generality) the last known value of the neighbor to the left. That neighbor can have a positive, negative, or zero value with respect to u_k. If the neighbor has a positive value, then the value of u_k goes up by one quantum size and the difference

between u_k and its neighbor is reduced. A symmetric argument applies when the left neighbor is negative with respect to u_k. We can conclude that the 1-norm is strictly decreasing as a function of the number of internal events.

Case 2 If two interior cell undergo a confluent event, say u_k and u_{k-1}, then, without loss of generality, u_k increases by one quantum and u_{k-1} decreases by one quantum for a net change of zero.

Now we only need show that the 1-norm must eventually decrease. Suppose that the wave is moving left. Consider the cell on the rightmost active portion of the wave. Since its neighbor is passive, if can only undergo an internal event. From Case 1, internal events can only decrease the magnitude of spatial derivative at that point (and, in doing so, decreases the 1-norm). The spatial derivative is an integer multiple of the quantum size and changes in the derivative occur at a rate of one quantum per internal event. It can be concluded that the spatial derivative must reach zero, and so the cell will become passive. Repeating this argument for each cell, moving from left to right, completes the proof.

9.1. Activity Tracking Property of the DEVS-Based Numerical Scheme

The Sod shock tube problem (see [Sod 1978]) involves the motion of shock and rarefaction waves along which the spatial derivative is large. The wave fronts are separated by stable plateaus where the spatial derivative is very small. Computationally, the cells through which a wave front is moving will undergo frequent state changes since

the time advance function for those cells will take on a small value. The cells along the stable plateaus will have very large, possibly infinite, time advances and so undergo only infrequent state changes. This will result in the discrete event simulation focusing its computational effort on the wave front, while locally stable regions receive relatively little attention. As the wave moves through space, the computational focus shifts accordingly. This front tracking behavior is an intrinsic property of the discrete event system.

The effect of this front tracking behavior on the simulator execution time is substantial. To illustrate the effect on a single computer, the discrete event model and corresponding discrete time model were implemented in C++ and executed on a desktop computer with a 500 MHz AMD processor and 198M RAM running the Linux operating system. The discrete time model uses explicit Euler to integrate through time. The same iterative flux approximation function was used for both implementations (thanks to Andrei A. Chernousov at http://www.geocities.com/andrei_chernousov/ for providing this code).

The Sod shock tube problem was solved using the same discretization in space for both the explicit Euler and quantized state integration schemes. A fixed quantum size was used for the discrete event solution. The step size for the discrete time solution was taken to be the smallest value of the time advance function computed as a result of an internal or confluent event. Figure 9.2 shows the solution computed at times 0.0 (initial conditions), 0.1, and 0.2.

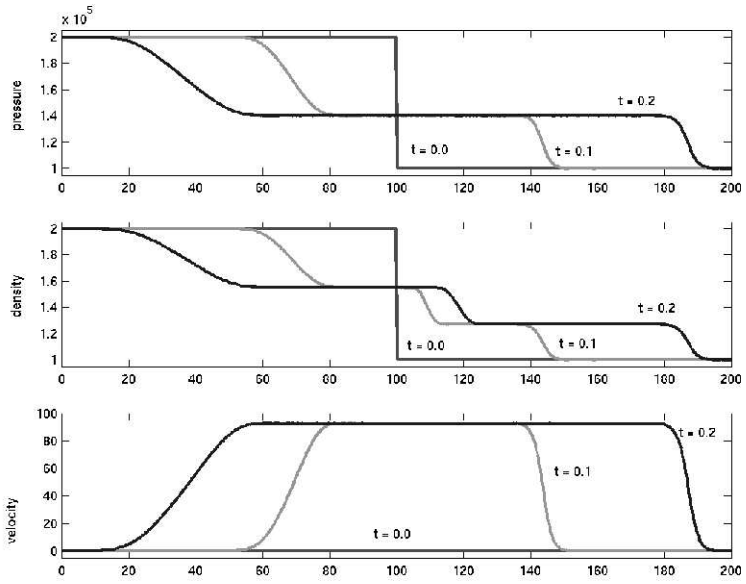

Figure 9.2. Computed wave fronts for the Sod shock tube problem.

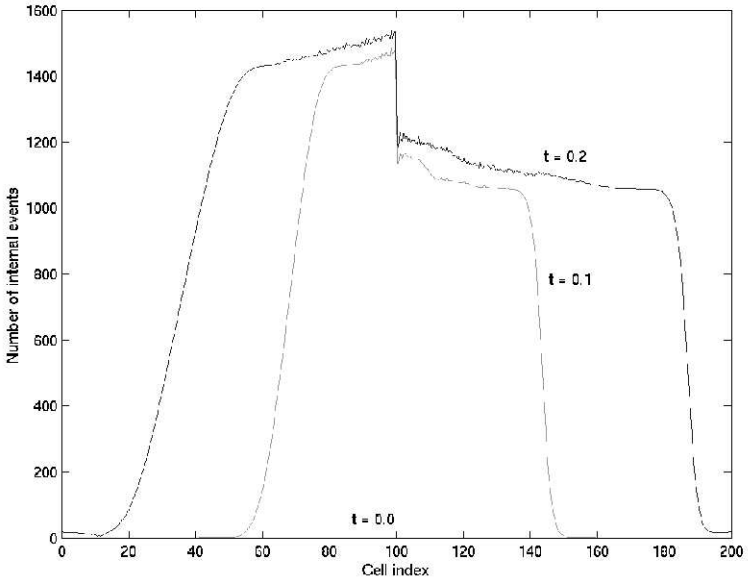

Figure 9.3. Number of internal state changes required to compute the wave fronts depicted in Figure 9.2.

The allocation of computational effort for the discrete event solution is shown in Figure 9.3. It can be seen that the discrete event solution focuses computational effort on the active portions of the solution as the shockwave moves outward. Also notice that regions untouched by the wave have not received any attention at all.

Table 9.1 shows the parameters used for each choice of discretization in space and the resulting time to solution. It can be seen that the advantage of the DEVS solution, relative to the discrete time solution, increases as the number of cells grows from 200 to 1600. At 1600 cells, the speedup of the DEVS solution appears to level off at a value of about 34. The combination of focused computational effort and significantly reduced number of computations per cell has produced more than an order of magnitude improvement in the time to solution.

Table 9.1. Solution parameters and speedup for DEVS and discrete time (DTSS) techniques.

Cells	Quantum size (U1,U2,U3)	Time step	DEVS soln. time (seconds)	DTSS soln. time (seconds)	Relative speedup of DEVS
200	(0.1,1.0,1000.0)	1.87E-005	2.23	19.01	8.52
1000	"	3.74E-006	13.37	380.76	28.5
1600	"	2.34E-006	26.95	941.73	34.9
2000	"	1.87E-006	41.19	1429.42	34.7
4000	"	9.35E-007	165.73	5482	32.8
6000	"	6.23E-007	339.55	10534.5	31.0
8000	'	4.67E-007	622.94	21585	34.7

The relative speedup of the discrete event solution seems to approach a constant value of roughly 34. This is not the expected relative speedup which might be anticipated by considering the activity of the solution [Jammalamadaka 2003]. First, notice that the

numerical solution has sharp derivatives along the wave front and near zero derivatives everywhere else. One consequence of this is that the magnitude of the difference between the initial and final (i.e., end of simulation time) value at each cell is roughly a constant or it is zero. This is illustrated in Figure 9.4 which shows $|\mathbf{p}(0) - \mathbf{p}(0.2)|$, where $\mathbf{p}(t)$ is the vector of primitive state variables used to compute the solution depicted in Figure 9.2.

We can roughly approximate the relative speedup of the discrete event solution. Let k be the mean, absolute difference between the initial and final state of each cell. The number of state changes required to achieve that difference using the DAB1 integrator with quantum size q is k/q. If we assume that all N cells makes that number of state changes, then the number of state changes that must be computed to complete the discrete event simulation is given by Nk/q. Let T be the number of time steps required for the discrete time simulation to generate its solution. Then the number of state changes computed by the discrete time simulation is equal to NT.

The relative speedup of the discrete event to discrete time solution can be approximated as Tq/k. The explicit Euler and DAB1 methods exhibit errors proportional to their integration parameters. It seems reasonable to assume that, in order to maintain similar errors with both methods, T must be inversely proportional to q. Letting $T = \theta/q$ gives a relative speedup of θ/k which, as it will be shown, is linearly dependent on the magnitude of the first derivative (i.e., the magnitude of the flux across a cell boundary).

If we set as equal the truncation error at the n-th step for each integration scheme then

$$9\text{-}7) \qquad \frac{h^2 \ddot{Y}(\zeta_n)}{2} = \frac{q^2 \ddot{Y}(\nu_n)}{2|\dot{y}_n|^2} \quad .$$

Rearranging and multiplying by the simulation completion time t_{end} gives

$$9\text{-}8) \qquad \frac{t_{end}}{h} = T = \frac{|\dot{y}_n| t_{end}}{q} \sqrt{\frac{\ddot{Y}(\zeta_n)}{\ddot{Y}(\nu_n)}} \quad ,$$

so

$$9\text{-}9) \qquad \theta = |\dot{y}_n| t_{end} \sqrt{\frac{\ddot{Y}(\zeta_n)}{\ddot{Y}(\nu_n)}} \quad .$$

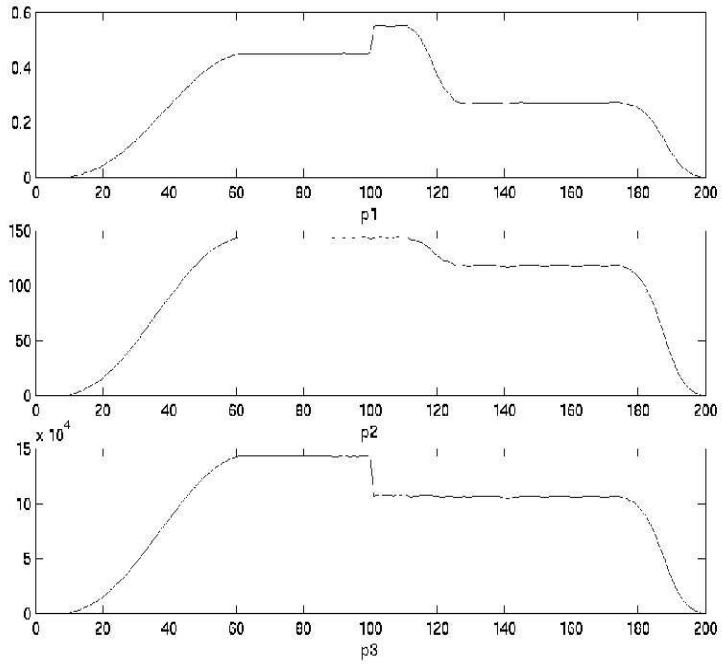

Figure 9.4. Difference between the final and initial state of the shock tube for a 0.2 second simulation using 200 cells.

Table 9.2 shows the anticipated relative speedup of the discrete event scheme calculated using 1/3 θ/k and 1/3 Tq/k, where T is computed directly using t_{end}/h. In both cases, k is take to be 12×10^4. The factor of 1/3 accounts for the overhead per state transition that appears in the discrete event simulation (see [Jammalamadaka 2003]). When computing θ, the largest derivative observed during the discrete event simulation is used. As can be seen from the table, the expected speedup of the discrete event scheme relative to the discrete time scheme grows linearly in the number of cells.

Table 9.2. Anticipated speedup of the discrete event simulation with respect to the discrete time simulation.

Cells	Maximm derivative	1/3 θ/k	1/3 Tq/k	Observed relative speedup of DEVS
200	4.60E+007	26	30	8.52
1000	2.30E+008	127	149	28.5
1600	3.70E+008	207	237	34.9
2000	4.60E+008	257	297	34.7
4000	9.20E+008	510	594	32.8
6000	1.40E+009	777	892	31.0
8000	1.80E+009	1000	1190	34.7

From the data in Table 9.2, it is tempting to conclude that linear speedup can be achieved by the discrete event solution if only we could somehow improve the simulation engine. Unfortunately, this is not the case. It can be determined empirically that the ratio of state changes computed by the discrete event simulation to the number of state changes computed by the discrete time solution does, in fact, approach a constant as the number of cells is increased.

The speedup model just presented would be largely adequate if not for the oscillations inherent in the DAB1 integration scheme (see Chapter 8). The number of state transitions required to compute the discrete event solution is, according to our speedup model, is expected to grow as Nk/q where N is the number of cells. In practice, this is not the case. Rather, the number of state changes computed by the discrete event scheme grows non-linearly as the number of cells is increased. Table 9.3 shows the corrected anticipated speedup values that can be computed using a count of the state transitions computed by the discrete event simulation, rather than the approximation k/q.

Table 9.3. Corrected anticipated speedup of the discrete event simulation with respect to the discrete time simulation.

Cells	Total DEVS state transition count	Corrected 1/3 θ/k	Corrected 1/3 Tq/k	Observed relative speedup of DEVS
200	84092	7.7	8.5	8.52
1000	559676	28.6	32	28.5
1600	1197932	34.8	38	34.9
2000	1756828	36.8	41	34.7
4000	6265668	41	46	32.8
6000	14021468	42.3	46	31.0
8000	25041624	40.3	46	34.7

The unanticipated computational effort that appears as a result of the oscillations due to the DAB1 integration scheme accounts for much of the optimism that is apparent in the anticipated speedup shown in Table 9.2.

9.2. Parallel Simulation

If a solution to the Sod shock tube problem is attempted using Algorithm 7.3, it is expected that the simulation will separate into three relatively non-interacting tasks corresponding to the two wave fronts appearing in the solution and the density discontinuity that remains near the center. Since there are three distinct active regions, it is anticipated that the best possible speedup is 3. In contrast to this, a parallel implementation of the discrete time solution is expected to exhibit nearly linear speedup for a small numbers of processors, with the speedup approaching an upper bound as the numbers of processors becomes large.

To demonstrate the relatively performance of the parallel discrete event and parallel discrete time solutions, a 10000 cell case was executed on a Scyld Beowulf cluster. Equal sized blocks of contiguous cells were assigned to each processor. For the discrete event solution, the integration quantum was taken to be an order of magnitude smaller than listed in Table 9.1. The discrete time solution used the largest possible time step that ensured a stable computation. If the time step were selected as before, then it would be equal to 1.87E-8. However, the corresponding execution time prohibits the use such a small time step.

The discrete time solution was implemented using the following parallel algorithm. Each processor contains a block of cells numbered 1 to N. Cells 1 and N are ghost cells. A processor is responsible for updating the state of cells 2 to N-1. At the beginning of each time step, the processor sends the state of cells 2 and N-1 to its left and

right neighboring processors respectively. The processor then receives the state of cells 1 and N from at its left and right neighboring processors. The processor then updates the states of cells 2 to N-1. These steps are repeated until the simulation end time has been reached.

Table 9.4 shows the speedup of the discrete time and discrete event simulations as a function of the number of processors used in the computation. It can be seen that the discrete event solution achieves a maximum speedup of 2.5, which is 83% of the anticipated maximum speedup. The discrete time solution reaches a speedup of 8.1 using seventeen processors. This is 3.2 times the speedup achieved by the discrete event algorithm. If we assume that, on a single processor, the discrete event simulation is at least 30 times faster than the discrete time solution, then the net speedup of the parallel discrete event solution with respect to the parallel discrete time solution is at least 9. If we further assume that the speedup of the parallel discrete time solution will remain linear with a slope of one half, then at least 18 more processors will be needed before the parallel discrete time solution can be computed as rapidly as the parallel discrete event solution. This last assumption is very optimistic, and it is unlikely to be realized without specialized hardware.

Table 9.4. Speedup of the parallel discrete event and discrete time algorithms.

Processors	DEVS speedup	DTSS speedup
1	1.0	1.0
2	1.2	1.8
3	1.4	2.4
4	1.8	2.9
5	2.0	3.4
6	2.2	4.0
7	2.3	4.4
8	2.5	4.8
9	2.3	5.3
10	2.4	5.7
11	1.3	6.1
12	0.5	6.1
13		6.4
14		6.6
15		6.7
16		7.6
17		8.1

10. THE WAVE EQUATION IN ONE DIMENSION

A one dimensional medium with varying capacitance and inductance (or bulk/shear modulus and density for physical mediums) can be modeled using equation

$$\text{10-1)} \qquad \frac{\partial^2 u}{\partial t^2} = \frac{1}{c}\frac{\partial}{\partial x}\left(\frac{1}{l}\frac{\partial u}{\partial x}\right) \quad .$$

This equation can be approximated using digital wave guide networks. When the parameters l and c are constant, the digital wave guide network provides an exact solution to (10-1). If l and c are allowed to be functions of x, then the digital wave guide network is equivalent to a second order finite difference approximation of (10-1) (see [Bilbao 2001]).

Unlike a second order finite difference approximation, the digital wave guide network can be expressed directly in terms of a coupled DEVS model. This leads to a more efficient computational scheme which focuses its computation effort on the moving waves. Moreover, a discrete event simulation of equation (10-1) can be very effectively parallelized using Algorithm 7.3.

10.1. Digital Wave Guide Networks

We will consider only a one dimensional digital wave guide network. For very thorough treatment of the subject see [Bilbao 2001]. Our goal is to model the propagation of a wave through a one dimensional medium whose wave speed and density

are a function of space. The compressibility of the medium is related to the wave speed and density by

$$10\text{-}2) \qquad c = \frac{1}{v^2 l} \quad .$$

The medium can be divided into equal sized cells which are internally homogenous. Suppose a particular cell has width dx and speed v. If a wave enters a cell from the left (right) it will leave the cell at right (left) dx/v units of time later. If the cell and its neighbor have equal speed and density, then we have already arrived at a solution. Let u_{+l} and u_{+r} be the waves entering the cell through the left and right boundaries and let u_{-l} and u_{-r} be the waves leaving the cell through its left and right boundaries. Then at the cell boundaries we have

$$10\text{-}3) \quad u_{-l} = u_{+r} \text{ and}$$

$$u_{-r} = u_{+l}.$$

More generally, if we want to consider an initial value at the cell we can formulate the input-output model as

$$U_j = u_{+l} + u_{+r},$$

$$10\text{-}4) \quad u_{-l} = U_j - u_{+l}, \text{ and}$$

$$u_{-r} = U_j - u_{-r}$$

where U_j is used to described the initial displacement of the medium. A discrete time simulation of this wave consists of computing U_j at every cell each time step, where a single time step corresponds to dx/v units of time. A discrete event variant of this

model can be readily obtained by setting the time advance to infinity if U_j, u_{-l}, and u_{-r} are zero and only exchanging non-zero outputs with the left and right neighboring cells.

This simple model can be augmented to accommodate spatial variations in the wave speed and medium density. The augmented model has the same form as (10-4), but uses a wave impedance at the cell boundaries to compute reflected and transmitted energy. A feedback loop is used to control the release of energy when the wave speed on the left and right sides of a boundary are not equal. This new model is derived in [Bilbao 2001] and is described by

$$\begin{aligned} U_j^{n+1} &= \frac{2(U_{+l}Z_l + U_{+r}Z_r + d^n Z_c)}{Z_l + Z_r + Z_c} \\ d^{n+1} &= U_j^{n+1} - d^n \\ U_{-l} &= U_j^{n+1} - U_{+l} \\ U_{-r} &= U_j^{n+1} - U_{+r} \end{aligned} \quad , \tag{10-5}$$

where d represents the feedback (or delay) loop and the subscripted Z are the wave impedance on the left (Z_l) and right (Z_r) boundaries and along the feedback loop (Z_c). These values are given by

$$\begin{aligned} Z_l &= \frac{1}{v_{max}\, c(x - \frac{dx}{2})} \\ Z_r &= \frac{1}{v_{max}\, c(x + \frac{dx}{2})} \\ Z_c &= v_{max}\left(1(x - \frac{dx}{2}) + 1(x + \frac{dx}{2})\right) - \frac{1}{v_{max}}\left(\frac{1}{c(x - \frac{dx}{2})} + \frac{1}{c(x + \frac{dx}{2})}\right) \end{aligned} \quad , \tag{10-10}$$

where v_{max} is the largest wave speed that appears in the calculation and x is the position of the cell.

If the medium has a constant wave speed, then a simplified form of (10-10) can be derived from basic principles. Consider a wave interacting with the cell boundary depicted in Figure 10.1. The density of the cell that the wave is originating from is denoted by l_1. The density of the cell that the wave is entering is denoted by l_2.

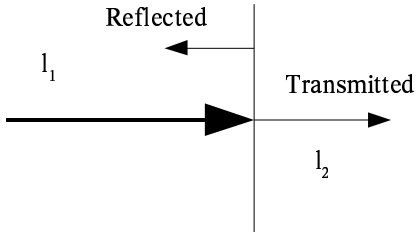

Figure 10.1. A wave encountering encountering a discontinuity in the medium.

The fraction of the wave that is transmitted through the cell boundary is given by

$$10\text{-}11) \quad \frac{2l_1}{l_1+l_2} \quad .$$

The fraction of the wave reflected from the cell boundary is given by

$$10\text{-}12) \quad \frac{l_2-l_1}{l_1+l_2} \quad .$$

At each time step, the wave leaving the left boundary of a cell is given by the fraction of the left moving wave that is transmitted through the left boundary plus the fraction of the right moving wave that was reflected from the right boundary. The wave leaving the right boundary is similarly computed. The input-output representation of a cell is given by

$$10\text{-}13)\qquad \begin{aligned} U_{-l} &= \frac{2l_r U_{+r} + (l_l - l_r)U_{+l}}{l_r + l_l} \\ U_{-r} &= \frac{2l_l U_{+l} + (l_r - l_l)U_{+r}}{l_r + l_l} \end{aligned} \quad .$$

Since the wave speed remains constant, we can simulate this model using the discrete time or discrete event scheme used in the homogeneous case. The discrete event scheme will have an advantage, however, since cells that do not receive input (i.e., that are not occupied by the wave) will have an infinite time advance and so will not consume any computational resources.

When the wave speed is allowed to vary, a discrete event simulation can be constructed using (10-10). Unfortunately, the delay element causes the space to be active everywhere that the wave has passed through. The discrete event scheme is, in that case, actually worse than the discrete time scheme since the scheduling and other overhead associated with the discrete event simulation become significant.

It would be interesting to consider a pure discrete event scheme in which (10-13) is used in regions of space where the wave speed is constant and wave compression/decompression is used to maintain the continuity of the wave at cells where the wave speed varies. In such a scheme, wave packets arriving at a variable speed junction would be combined or split, as needed, to ensure that the wave front remains continuous. Wave packets moving into a faster medium would leave the variable speed cell sooner than waves moving into the slower medium. In effect, a variable speed cell would act as a wave packet queue in which items entering the queue can be split/combined and items leaving the queue are assigned a 'service time' according to its

direction of travel. It may be possible to extend such a scheme to two dimensions using a variation of the voltage-centered 2D mesh described in [Bilbao 2001]. However, further development of these ideas is outside the scope of our current pursuit and so must be left for future research.

10.2. Solving the Wave Equation on a Parallel Computer

Consider a string 100π units in length with a wave speed of 1. In two separate experiments, we will use two strings each broken into equally sized regions with differing densities as shown in Figure 10.2. One string is homogenous and the other has three otherwise homogenous regions.

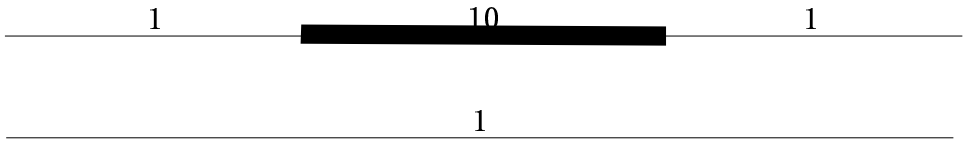

Figure 10.2. A homogenous (below) and inhomogeneous (above) string.

The string is initially plucked in the center giving it a displacement described by

$$10\text{-}14)\quad u(x) = \begin{cases} \sin(x - 99\pi/2) & \text{if } 99\pi/2 \le x \le 101\pi/2 \\ 0 & \text{otherwise.} \end{cases}$$

Figure 10.3 show solutions computed on each string using a digital wave guide network with 1000 cells.

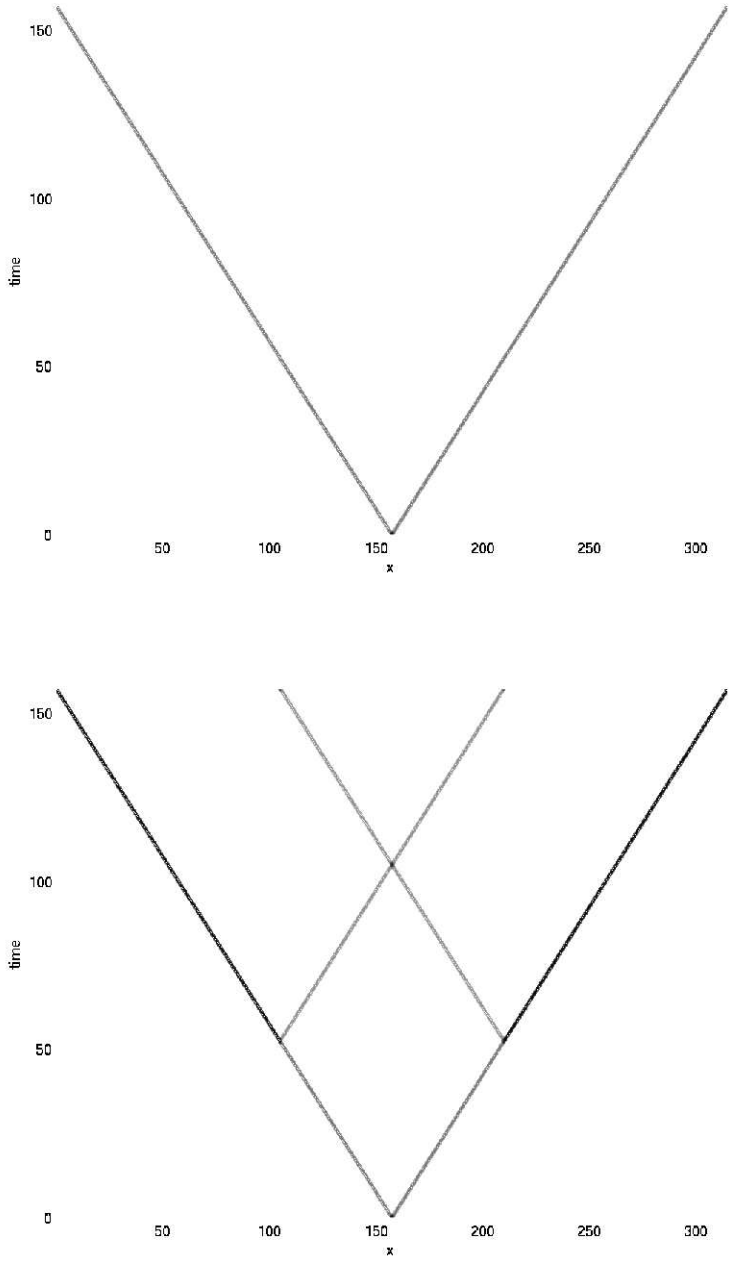

Figure 10.3. Solution to equation (10-1) solved on a homogenous string (above) and inhomogeneous string (below).

The above solution was computed using both Algorithm 7.3 and a parallel discrete time algorithm similar to that used in Chapter 9. The string is evenly divided amongst the available processors. Figures 10.4 and 10.5 show the time required to compute a solution using both algorithms with a given number of grid points and processors. Figure 10.4 shows the execution times for the homogenous string and Figure 10.5 shows execution times for the inhomogeneous string. In both cases the string was approximated by 50k cells. Figures 10.6 and 10.7 show the actual speedup obtained for homogenous and inhomogeneous problems, respectively.

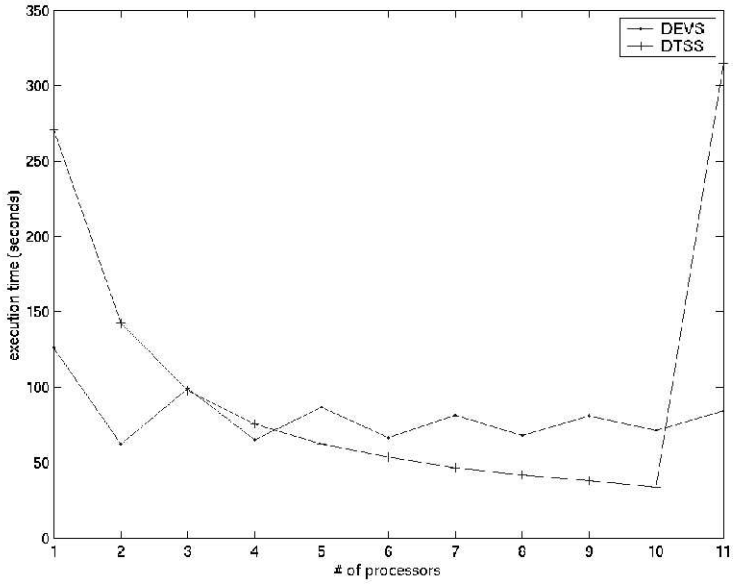

Figure 10.4. Execution times for the homogenous string problem with 50k cells.

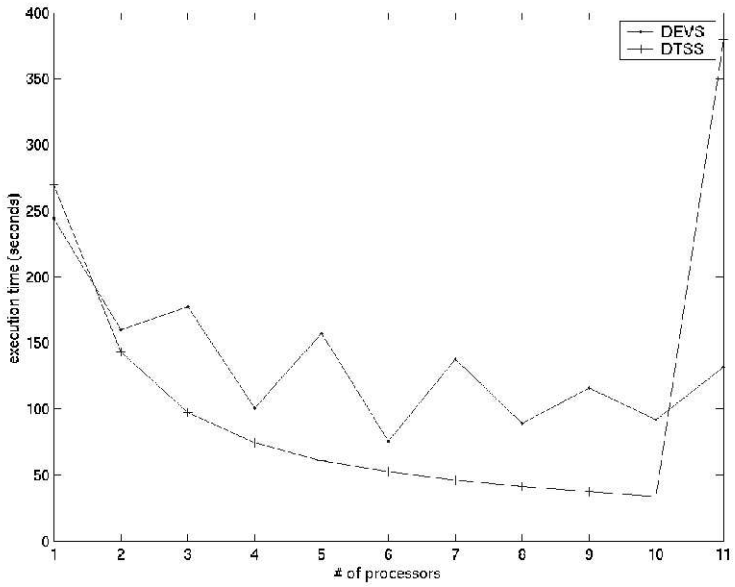

Figure 10.5. Execution times for the inhomogeneous string problem with 50k cells.

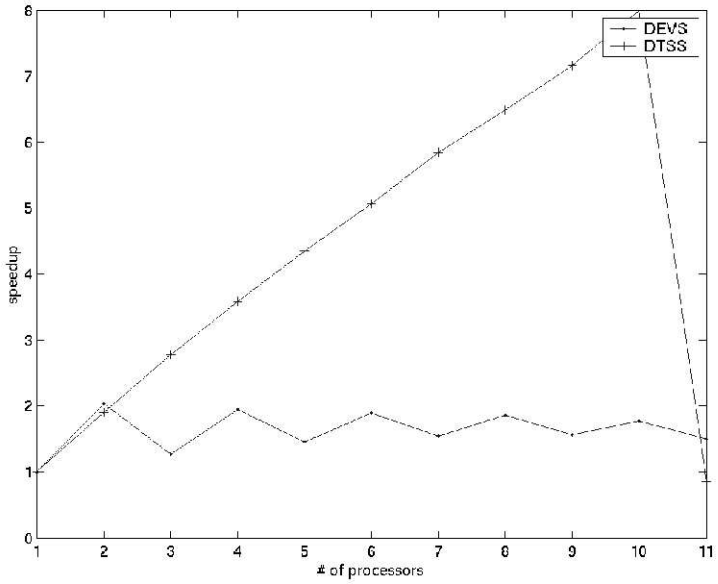

Figure 10.6. Speedup for the homogenous string problem with 50k cells.

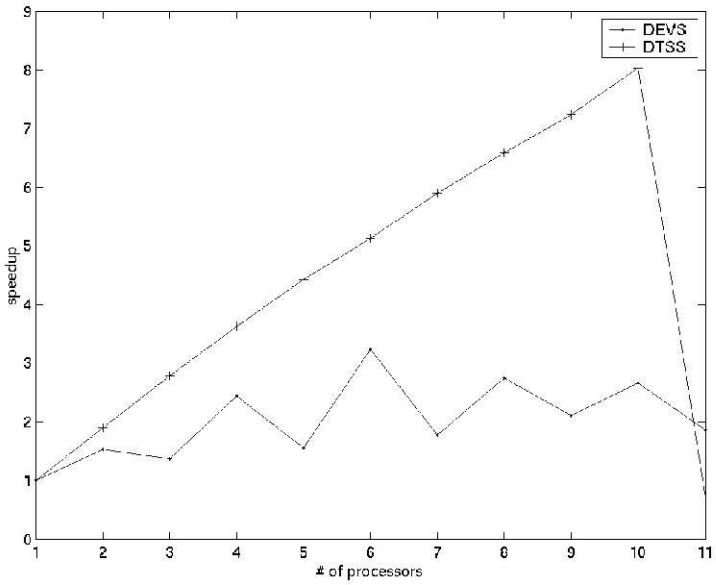

Figure 10.7. Speedup for the inhomogeneous string problem with 50k cells.

Figures 10.4 through 10.7 demonstrates the data dependent performance of the parallel discrete event algorithm. In contrast to this, the discrete time algorithm is insensitive to changes in the behavior of the wave. From the solutions shown in Figure 10.3, we would expect the discrete event algorithm to exhibit a peak speedup of 2 for the homogenous case and 4 in the inhomogeneous case. The actual speedup of the discrete event solution is 2 and 3.2 for the homogenous and inhomogeneous cases, respectively. Notice that the maximum speedup in the homogenous case coincides with a perfect partitioning of the problem. That is, the cell space is divided down the middle, with each travelling wave being simulated independently. It seems likely that more careful partitioning of the inhomogeneous problem could more nearly approach a perfect speedup number.

Algorithm 7.3 provides the expected speedup with respect to the discrete event simulation executed on a single computer. Initially, we might have hoped that it would provide a competitive alternative to the parallel discrete time solution. In the case of a homogenous string simulated using one and two processors, an overall speedup of 2.3 was achieved by Algorithm 7.3. However, this advantage eroded rapidly as more processing power was brought to bear. The single computer advantage that the discrete event simulation had over the discrete time solution was not adequate, in this case, to overcome the relative ease with which the discrete time solution could be parallelized.

Even the initial advantage of the discrete event solution disappeared when we attempted to simulate the heterogeneous string. This reduction in the initial advantage of the discrete event solution can be explained in terms of the ratio of active to inactive cells. In the homogenous case, the number of active cells was fixed at 1000, 500 moving left and 500 moving right. The total number of cells was 50000. The simulation required 25000 time steps to complete. The discrete time simulator computes the state of each cell at each time step, and so it computed 1250 million state changes. The discrete event solution computed only 25 million state changes.

When we simulated the inhomogeneous string, the number of active cells increased during the course of the simulation. From time step 0 to 9000 or so, there were 1000 active cells. This number doubled at about time step 9000. The number of state changes computed by the discrete time system remains the same as in the inhomogeneous case. However, the number of state changes computed by the discrete event simulation increases almost two times to about 41 million.

By the same argument, it is expected that if the number of active cells is reduced, then the discrete event scheme will show an advantage with respect to the discrete time scheme. This can be readily demonstrated by modifying the inhomogeneous string problem in two ways. First, consider the effect of halving the width of the initial pulse. That is, we set the initial conditions to be

$$u(x) = \sin(x-(100\pi/2-\pi/4)) \quad \text{if } 100\pi/2-\pi/4 \le x \le 100\pi/2+\pi/4$$

$$0 \quad \text{otherwise.}$$

This reduces the number of state changes computed by the discrete event scheme by two. The number of state changes computed by the discrete time scheme remains unchanged. Table 10.1 shows the corresponding execution time and speedup of the discrete event scheme in this case. The worse case relative speedup of the discrete event solution with respect to the discrete time solution in this case is 1.5.

Table 10.1. Execution time and speedup for the discrete event solution using narrow initial conditions.

Processors	1	2	3	4	5	6	7	8	9	10
Execution time(secs)	98	57	67	34	57	26	48	30	40	135
Speedup	1	1.7	1.5	2.9	1.7	3.8	2	3.3	2.5	0.7

The number of active cells can be further reduced by changing the density profile of the string. For example, consider a string 100π units in length for which the left two thirds of the string have a density of 10 and the right one third has a density of 1. The solution to equation (10-1) on this string using the narrow pulse just defined is shown in figure 10.8.

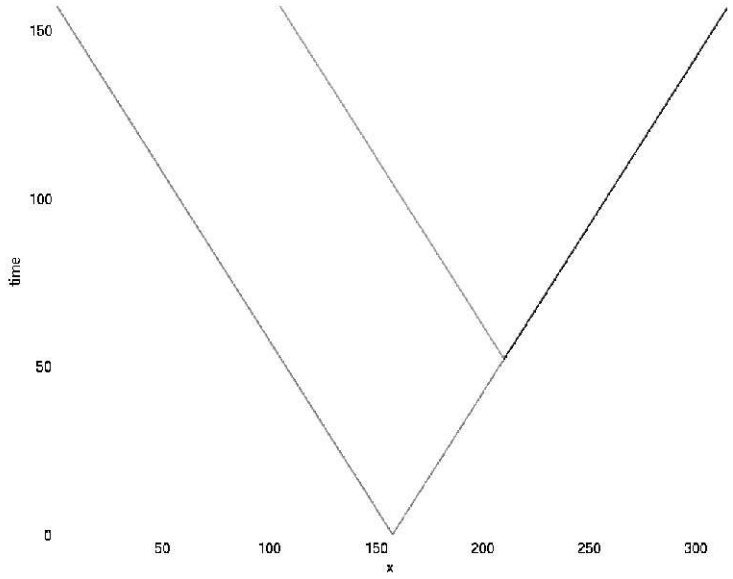

Figure 10.8. Solution to (10-1) using a narrow initial pulse with a string whose left two thirds and right one third have densities of 10 and 1, respectively.

By removing the left most discontinuity in the string density, the number of state changes computed by the discrete event system is reduced once more. In this case, the reduction is by roughly one third. A corresponding reduction in the time to solution is observed. The execution times and speedup of the discrete event solution are shown in table 10.2. In this case, the relative speedup of the discrete event solution with respect to the discrete time solution is 1.6.

Table 10.2. Execution time and speedup for the discrete event solution using narrow initial conditions and one less region of differing density.

Processors	1	2	3	4	5	6	7	8	9	10
Execution time(sec.)	67	57	38	33	39	24	35	29	29	142
Speedup	1	1.2	1.8	2	1.7	2.8	1.9	2.3	2.3	0.47

A relative reduction in effort for the discrete event scheme can be obtained in either case by increasing the distance travelled by the wave in the course of the simulation. Consider the effect of doubling the string length, with a corresponding doubling of the simulated time interval, for the inhomogeneous string simulation shown in Figure 10.3. The motion of the wave was simulated on a string of length 400π and 800π with a spatial resolution of 0.002π. This resolution requires 200000 cells to simulate the wave moving on the shorter string and 400000 cells for the longer string. The initial conditions are those given in (10-14). Table 10.3 shows the execution times of the discrete time and discrete event schemes as a function of the number of processors used to run the simulation.

Table 10.3. Execution time for the discrete time and discrete event simulation of the short and long string problems as a function of the number of processors used in the simulation.

Processors	Short string (DEVS)	Short string (DTSS)	Long string (DEVS)	Long string (DTSS)
1	745 seconds	4324 seconds	1475 seconds	17249 seconds
2	529	2230	1049	8858
3	613	1520	1227	5970
4	327	1144	631	4536
5	538	927	1051	3606
6	231	759	451	2947
7	453	658	893	2592
8	278	587	547	2288
9	365	516	724	2045
10	455	486	536	1848

The execution time of the discrete time scheme grows four times when the string length is doubled. This corresponds to a doubling both in the number of cells used in the simulation and the number of state changes computed at each cell. In general, the number of state changes computed by the discrete time scheme will be proportional to the number of cells squared.

In contrast to this, the discrete event scheme requires only twice as much time to complete the simulation when we move from the shorter to the long string. The simulation execution is only doubled in this case because we simulate twice as many state transitions, but the number of active cells (i.e., the number of cells used to approximate the pulse) remains the same. In general, the number of state changes will be proportional to the number of cells. The relative speedup of the discrete event scheme with respect to the discrete time simulation is expected to grow linearly with the number of cells.

When simulating this problem on a parallel computer, the parallel discrete event scheme continues to outperform the parallel discrete time scheme. Table 10.4 shows the relative speedup of the parallel discrete event simulation with respect to the parallel discrete time simulation.

The shortest execution times are achieved with the parallel discrete time scheme using 6 processors. If the discrete time scheme and discrete event scheme are given the same number of processors, then the relative speedup of the discrete event scheme is 3.3 for the short string simulation and 6.5 for the long string simulation. If we assume the that slope of the speedup curve for the parallel discrete time scheme remains at a 0.9, then

we would need 14 processors (i.e., a total of 24 processors) before the parallel discrete time scheme could be executed as quickly as the parallel discrete event scheme.

Table 10.4. Relative speedup of the discrete event scheme with respect to the discrete time scheme as a function of the number of processors used to execute the simulation.

Processors	Short string	Long string
1	5.8	11.7
2	4.2	8.4
3	2.5	4.9
4	3.5	7.2
5	1.7	3.4
6	3.3	6.5
7	1.5	2.9
8	2.1	4.2
9	1.4	2.8
10	1.1	3.4

11. CONCLUSIONS

Discrete event simulation has traditionally focused on networks of interacting logical processes. Within this framework, parallel discrete event simulation has been developed to enable faster and more memory efficient computations. Unfortunately, the lack of a well define modeling framework has made it nearly impossible to determine if a particular PDES algorithm will be able to simulate a given system. In order for parallel discrete event simulation to be adopted by a wider community requires that proposed algorithms be demonstrably correct. Only when we know that an algorithm will work does it become meaningful to look for improved performance.

By shifting to a systems oriented view point, we are able to develop algorithms for which the term 'correct' has a clearly defined meaning. The algorithms developed in this dissertation correctly simulate the class of systems that can be represented in the DEVS formalism. Consequently, they do not exhibit any of the traits that plague many logical process based simulation engines. In particular, the algorithms provide perfect repeatability in the sense that, given a system begun in some initial state, the algorithm will always produce the same answer. Moreover, that answer will be exactly the one dictated by the systems formalism.

Having developed these discrete event simulation algorithms, it is natural to ask where they might be useful. At the same time, we would like to demonstrate the unique capability of these algorithms to simulate DEVS models that are very sensitive to simulation errors. Numerical solutions to PDEs answer both of the needs.

One issue that arises when trying to solve partial differential equations is the prevalence of short feedback cycles. Whereas the Time Warp algorithm is generally thought to be the most promising PDES algorithm for general purpose use, it is clearly unsuited for simulating DEVS approximations of large, continuous systems. Adaptive variants of the Time Warp algorithm have been proposed as a means of coping with rollback and anti-message explosions, the former occurring in PDE simulations. It is an open question as to whether adaptive Time Warp algorithms can be successfully applied to continuous system simulation.

In solving the wave equation it was discovered that the discrete event simulation algorithm, even on a single computer, has considerably more overhead than its discrete time counterpart. This highlights the need for very efficient, reusable simulation engines. None the less, when the cost of computing a state change at a single cell is very high (e.g., in the shock tube problem) or the number of actual state changes are small with respect to the number of cells in the entire space (e.g., in the wave simulation) a significant reduction in time to solution can be achieved.

In this dissertation, discrete event simulations were compared against very simple, and non-adaptive, discrete time simulations. It would be interesting to compare the discrete event approach to other adaptive techniques (e.g., Adaptive Mesh Refinement). It is tempting to speculate that a discrete event technique would perform very well given a problem for which some adaptive step size or adaptive grid size technique provides a performance advantage.

It should also be noted that a substantial amount of research has been devoted to developing special purpose hardware to support classical methods in numerical analysis. Vector processors and systolic arrays provide excellent examples. Memory access patterns in discrete event simulations are likely to be less regular than those in 'for loop' based simulation programs. This last is particularly relevant when using commodity processors and compilers that provide built in support for software that exhibits 'locality of reference' in its memory access patterns. Many of the potential advantages of discrete event methods in the area of continuous systems simulations may require better hardware support before they can fully realized in practice.

Two other issues that were touched on briefly, but may be worth further consideration, are higher order discrete event integration methods and solutions to the wave equation via discrete event approximations. Higher order integration methods may be obtainable by using interpolating polynomials to construct more accurate time advance functions. This was demonstrated for a second order method which used linear interpolation to achieve second order accuracy. Even in this case, though, the question of how to extend the convergence proof to a system of equations remains open.

The solution to the wave equation using digital wave guides is another topic for future research. While it is possible to exactly represent a discrete time digital wave guide network as a coupled DEVS model, it may be possible to construct an energy conserving wave guide network by taking advantage of the DEVS time advance function. There is anecdotal evidence that this can be accomplished using wave packet queues to model the motion of a wave through an interface boundary. If such a technique could be

a developed, then the question of relative performance would take second seat to the improvement in accuracy. Practical applications of the wave equation are easy to find (e.g., in oil exploration, synthetic music, electro-magnetics, and optics), making this an attractive subject for future study.

REFERENCES

[Bagrodia 1991] Bagrodia, R., K. M. Chandy, Wen Toh Liao. "A Unifying Framework for Distributed Simulation." ACM Transactions on Modeling and Computer Simulation, vol. 1, no. 4, pp 348-385, October 1991.

[Barz 2003] Barz, Christopher, Rolf Göpffarth, Peter Martini, Andre Wenzel. "A new Framework for the Analysis of Simultaneous Events". Proc. of SCSC'03 (Summer Computer Simulation Conference), Montreal, Canada, July 2003.

[Bass 1997] Bass, L., P. Clements and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1997.

[Bilbao 2001] Stefan Damian Bilbao. *Wave and Scattering Methods for the Numerical Integration of Partial Differential Equations*. Diss. Stanford University, 2001.

[Bruce 1995] Bruce, D. "The Treatment of State in Optimistic Systems". Proceedings of the 9th Workshop on Parallel and Distributed Simulation, June 1995.

[Carothers 2000] Carothers, C. D, R. M. Fujimoto. "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms". IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 3, pp 299-317, March 2000.

[Cellier 1991]. Cellier, François E. *Continuous System Modeling*. New York: Springer-Verlag, 1991.

[Chow 1994] Chow, Alex ChungHen, Bernard P. Zeigler, Doo Hwan Kim. "Abstract Simulator for the Parallel DEVS Formalism". AIS 1994 Conference Proceedings, 1994.

[Christensen 1990] Christensen, E.R., *Hierarchical Optimistic Distributed Simulation: Combining DEVS and Time Warp*. Diss. University of Arizona, 1990.

[Christopher 1998] Christopher, H. Young, Nael B. Abu-Ghazaleh, Philip A. Wilsey. *OFC: A Distributed Fossil-Collection Algorithm for Time-Warp*. Lecture Notes in Computer Science. Ed. S. Kutten. Heidelberg: Springer-Verlag, January 1998.

[Damitio 1994] M. Damitio, S.J. Turner, C.J.M. Booth, M.J. Kirton, K.R. Milner and P.R. Hoare. "Comparing the Breathing Time Buckets Algorithm and the Time Warp Operating System on a Transputer Architecture". Proceedings of SCS European Simulation Multiconference, June 1994.

[Das 2000] Das, S. R. “Adaptive Protocols for Parallel Discrete Event Simulation”. Journal of the Operational Research Society (JORS), vol. 51, no. 4, pp. 385-394, April 2000.

[Deelman 1996] Deelman, E., T. Caraco, K.K. Szymanski. "Parallel Discrete Event Simulation of Lyme Disease”. Proceedings of the 1996 Pacific Symposium, World Scientific Publishing Co., 1996.

[Dickens 1990] Dickens, P., P. Reynolds, Jr. “SRADS with local rollback”. Distributed Simulation, vol. 22, pp. 161-164, Jan. 1990.

[Ferscha 1995] Ferscha, Alois. “Parallel and Distributed Simulation of Discrete Event Systems”. Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995

[Ferscha 1999] Ferscha, Alois. “Adaptive Time Warp Simulation of Timed Petri Nets”. IEEE Transactions on Software Engineering, vol. 25, no. 2, 1999.

[Frey 2002] Frey, Peter, Radharamanan Radhakrishnan, Harold W. Carter, Philip A. Wilsey, Perry Alexander. “A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation.” IEEE Transactions on Software Engineering, vol. 28, no. 1, pp 58-78, January 2002.

[Fujimoto 1997] Fujimoto, Richard, Maria Hybinette. “Computing Global Virtual Time in Shared-Memory Multiprocessors”. ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 4, pp 425-446, 1997.

[Fujimoto 1998] Fujimoto, Richard M. “Parallel and Distributed Simulation”. *Handbook of Simulation*. Ed. Jerry Banks. John Wiley and Sons, Inc. 1998.

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

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

[Ghosh 1996] Ghosh, Sumit. “On the Proof of Correctness of 'Yet Another Asynchronous Distributed Discrete Event Simulation Algorithm (YADDES)' ”. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 26, no. 1, pp 68-80, January 1996.

[Gray 1997] Gray, John Shapley. *Interprocess Communications in Unix: The Nooks & Crannies*. Upper Saddle River: Prentice Hall PTR. 1997.

[Hwang 1993] Hwang, Kai. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill Inc. 1993.

[Jammalamadaka 2003] Jammalamadaka , Rajanikanth. “Activity Characterization of Spatial Models: Application to Discrete Event Solution of Partial Differential Equations”. Master Thesis, University of Arizona, 2003.

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

[Jha 2000] Jha, Vikas, Rajive Bagrodia. “Simultaneous Events and Lookahead in Simulation Protocols”. ACM Transactions on Modeling and Computer Simulation, Vol. 10, No. 3, pp 241-267, 2000.

[Jefferson 1985] Jefferson, D. “Virtual Time”. ACM Transactions on Programming Languages and Systems, vol. 17, no. 3, pp 404-425, 1985.

[Karnopp 1975] Karnopp, Dean. *System Dynamics: A Unified Approach*. New York, Wiley, 1975.

[Klir 1969] George J. Klir. *An Approach to General Systems Theory*. New York: Van Nostrand Reinhold Co. 1969.

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

[Kofman 2003] Kofman, E. “Discrete Event Based Simulation and Control of Continuous Systems.” Diss. Universidad Nacional de Rosario, 2003.

[Lamport 1978] Lamport, Leslie. “Time, Clocks and the Ordering of Events in a Distributed System”. Communications of the ACM, vol. 21, no. 7, pp. 558-565, July 1978.

[Leivent 1993] Leivent, Jonathan I., Ronald J. Watro. “Mathematical Foundations in Time Warp Systems”. ACM Transactions on Programming Languages and Systems, vol. 15, no. 5, November 1993.

[Leveque 1996] Randall J. Leveque. “High-Resolution Conservative Algorithms for Advection in Incompressible Flow”. SIAM Journal of Numerical Analysis, vol. 33, no. 2, pp. 627-665, April 1996.

[Liao 1993] 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, September 1993.

[Lowry 2000] Lowry, Matthew C., Peter J. Ashenden and Ken A.Hawick. *A Testbed System for Time Warp in Java.* DHPC Technical Report DHPC-087, March 2000.

[Martin 1996] D.E. Martin, T. J. McBrayer, and P. A. Wilsey. "warped: A Time Warp Simulation Kernel for Analysis and Application Desvelopment". 29th Hawaii International Conference on System Sciences (HICSS-29), H. El-Rewini and B. D. Shriver (eds), Volume I, 383-386, January 1996.

[Mascarenhas 1995] Mascarenhas, Edward, Felipe Knop, Vernon Rego. "ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads". Proceedings of the 1995 Winter Simulation Conference, 1995.

[Mattern 1993] Mattern, F. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation". Journal of Parallel and Distributed Computing, 18: 423-434, 1993.

[Mattern 1999] Mattern, F. "Logical Time". Encyclopedia of Distributed Computing. Eds. P. Dasgupta, J. Urban. Kluwer Academic Publishers, 1999.

[Mesarovic 1989] Mesarovic, M.D., Y. Takahara. *Abstract Systems Theory.* Ed. M. Thomas, A. Wyner. Berlin: Springer-Verlag, 1989.

[Misra 1986] Misra, Jayadev. "Distributed Discrete-Event Simulation". Computing Surveys, vol. 18, no. 1, March 1986.

[Nicol 2000] Nicol, David, Jason Liu, James Cowie. "Safe Timestamps and Large-Scale Modeling". Proceedings of the 14th Workshop on Parallel and Distributed Simulation, May 2000.

[Page 1994] Page, Ernest H., Richard E. Nance. "Parallel Discrete Event Simulation: A Modeling Methodological Perspective." Proceedings of the ACM/IEEE/SCS 8th Workshop on Parallel and Distributed Simulation, pp 88-93, July 1994.

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

[Rajan 1995] Rajan, R., P.A. Wilsey. "Dynamically switching between lazy and aggressive cancellation in a Time Warp parallel simulator". Proceedings of the 28th Annual Simulation Symposium, April 1995.

[Ralston 2001] Ralston, Anthony, Philip Rabinowitz. *A First Course in Numerical Analysis*. 1978. Mineola: Dover Publications Inc. 2001.

[Rao 1998] Rao, D. M., N. V. Thondugulam, R. Radhakrishnan, P. A. Wilsey. "Unsynchronized Parallel Discrete Event Simulation". Proceedings of the 1998 Winter Simulation Conference (WSC'98), December 1998.

[Reed 1987] Reed, D.A., R. M. Fujimoto. *Multicomputer Networks: Message-based parallel processing*. Cambridge: MIT Press. 1987.

[Reynolds 1988] Reynolds, P.F. "A spectrum of options for parallel simulation". Proceedings of the Winter Simulation Conference, pp 352-332, 1988.

[Rönngren 1999] Rönngren, R., M. Liljenstam. "On Event Ordering in Parallel Discrete Event Simulation". Thirteenth Workshop on Parallel and Distributed Simulation, pp 38-45, 1999.

[Sarjoughian 2000] 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.

[Sarjoughian 2001] Sarjoughian, H.S., B.P. Zeigler, S.B. Hall. "A Layered Modeling and Simulation Architecture for Agent-Based System Development". IEEE Proceedings, vol. 89, no. 2, pp. 201-213, 2001.

[Snir 1998] Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. *MPI – The Complete Reference: Volume 1, The MPI Core, second edition*. Cambridge: MIT Press. 1998.

[Steinman 1992] Steinman, Jeff 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.

[Sod 1978] Sod, Gary A. "A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws". Journal of Computational Physics, vol. 27, pp. 1-31, 1978.

[Soliman 1999] Hussam M. Soliman, "On the Selection of the State Saving Strategy in Time Warp Parallel Simulations", Transactions of the Society for Computer Simulation International, Vol. 16, No. 1, pp. 32-36, 1999

[Tel 2000] Tel, Gerard. *Introduction to Distributed Algorithms.* Cambridge University Press, 2000.

[Thondugulam 1999] Thondugulam, Narayanan V., Dhananjai Madhava Rao, Radharamanan Radhakrishnan, Philip A. Wilsey. "Relaxing Causal Constrains in PDES". 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999.

[Tomlinson 1993] Tomlinson, Alexander I., Vijay K. Garg. "An Algorithm for Minimally Latent Global Virtual Time". Proceedings of the 7th Workshop on Parallel and Distributed Simulation, May 1993.

[Wieland 1999] Wieland, Frederick. "The Threshold of Event Simultaneity". Transactions of the Society for Computer Simulation International, vol. 16, no. 1, 1999.

[Wolfgang 1994] Wolfgang, K. Giloi. *Parallel Supercomputer Architectures and Their Programming Models*. Parallel Computing, Vol. 20, pp 1443-1470, 1994.

[Wymore 1993] Wymore, Wayne A. *Model-based systems engineering : an introduction to the mathematical theory of discrete systems and to the tricotyledon theory of system design*. Boca Raton: CRS Press, 1993.

[Zeigler 1999] Zeigler, Bernard P., George Ball, Hyup Cho, J.S. Lee, Hessam Sarjoughian. "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions". Simulation Interoperability Workshop, March 1999.

[Zeigler 2000] Zeigler, Bernard P., Herbert Praehofer, Tag Gon Kim. *Theory of Modeling and Simulation: Second Edition*. San Diego: Academic Press, 2000.