Terminating Parallel Discrete Event Simulations

by

D. S. Richardson

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Marc Abrams, Chairman

Lenwood Heath

Richard Nance

May, 1991

Blacksburg, Virginia

Terminating Parallel Discrete Event Simulations

by D. S. Richardson

Committee Chairman: Marc Abrams
Computer Science

(ABSTRACT)

This thesis analyzes the simulation termination problem of implementing global termination conditions and collecting output measures in discrete event simulations. With regard to parallel simulations, this problem is inherently more difficult than the classic termination detection problem for two reasons. The first is that parallel simulation processes are often written as nonterminating; the second is that the decision to terminate can not be independently made by each process contributing to the simulation.

A specification of a solution to the termination problem is developed as a sequence of stepwise refinements using UNITY, and proofs are given to demonstrate that each refinement satisfies the preceding specification. Termination conditions are categorized based on stability (if a condition is stable, once it becomes true it will remain true at all future times) and illustrated using space-time diagrams. A discussion is presented of how to implement termination conditions that are a combination of stable and nonstable conditions.

This thesis makes two major contributions. The first is an algorithm to implement global termination conditions and to collect the corresponding output measures in discrete event simulation. The specifications and algorithm given in this thesis are architecture independent, and apply to sequential as well as synchronous and asynchronous parallel discrete event simulation algorithms. The second is the development of a generalized, formal framework in which to reason about simulation algorithms. The techniques used in this thesis to solve the simulation termination problem may be applied to solve other problems arising in parallel simulation.

ACKNOWLEDGEMENTS

I was extremely privileged to have Dr. Marc Abrams as my thesis advisor. When I went to Dr. Abrams with a problem I returned with a solution, or a method I could follow to find a solution. I would leave his office feeling better than when I arrived, which differs from all expectation when dealing with an authority figure. Also, the professor has a gift of finding *something* good about work presented to him; seeing at least one "good" or "yes" in a paper full of edits and corrections does wonders. Many thanks to the fastest e-mailer in the business, even if he did kept me in Blacksburg working on this thesis about a month longer that I wanted to!

Thanks to Dr. Dick Nance and Dr. Lenny Heath for reading this thesis and offering their experience and helpful insight.

For invaluable financial and moral support, thanks to my parents. Won't you feel lonely without the standard end-of-the-month call for money? I guess I could continue that...

Lastly, thanks to the guys in the bridge fivesome and all my grad school friends, whose comaradarie and helpfulness made coming to school a pleasure.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

This thesis analyzes the problem of implementing global termination conditions and collecting output measures in discrete event simulations. With regard to parallel simulations, this problem is inherently difficult due to asychronous computation, lack of global knowledge, and because simulation processes are written as nonterminating. Previous parallel simulation research has not been concerned with implementing global termination conditions; local termination conditions have been used exclusively. Local termination conditions (e.g., terminating when each processor reaches a certain simulation time, terminating when each processor has processed a specified number of jobs) can be evaluated independently by each process using only private data. This thesis proposes methods to terminate simulations when the application of local conditions are undesirable or impossible.

To illustrate the impossibility of implementing a global termination condition using existing parallel simulation techniques, suppose that the purpose of a simulation program is to compute the value of some output measure after two processors in a parallel simulation have processed a total of X jobs. Parallel discrete event simulations often execute asynchronously, so one processor is likely to reach a certain point in simulation time before the other. Thus, once both processors have collected data for that simulation time, and it is determined that X messages have been processed, the processor which first reached that simulation time will have calculated beyond it.

Two key problems in implementing global termination conditions and collecting output measures are solved in this thesis. First, how can each of the (possibly) asynchronous processes comprising a simulation be brought to a halt once the termination

time has been determined? Second, how can output measures be obtained from a time in the past of the parallel simulation? These problems do not arise when a local termination condition is used because the prespecified termination point is designed for collecting the output measures.

1.1 IMPORTANT DEFINITIONS

This section defines terms that must be understood to discuss the thesis problem.

A *simulation program* is a representation of a *simulation model*. Nance [1981] explains:

> "A model of a system is comprised of *objects* and *relationships* among objects. An object is anything that can be characterized by one or more *attributes* to which *values* are assigned."

Regarding *simulation time*, Nance [1981] describes:

> "Every simulation model must have an indexing attribute, that is, an attribute of the model object that enables state transitions. Time is the most common indexing attribute..."

The *attributes* of a simulation model represent values held by the simulation system. The value of an attribute a changes at a sequence of discrete simulation times $t_1, t_2, \ldots$. Therefore the value of an attribute remains the same in time interval $[0,t_1)$, then assumes a new value and remains the same in interval $[t_1, t_2)$, and so on. The *value of an attribute at simulation time t* is the value the attribute was assigned at the largest $t_i \leq t$.

A *parallel simulation* program is a mapping of a simulation program to a set of processors. In a parallel simulation program, each attribute is modified by (or is local to) exactly one processor.

A *termination condition* specifies a relationship among the values of some subset of simulation model attributes. Evaluation of a termination condition yields one of the values "true" or "false", indicating that the condition is or is not satisfied.

A *local termination condition* is a termination condition that can be evaluated by a processor using only data that is private to that processor. Examples of local termination conditions include "terminate once this processor has reached simulation time T" or "terminate once X jobs have been completed by this processor."

A *global termination condition* must be evaluated with knowledge of data from more than one processor, typically from all processors contributing to the simulation. An example of a global termination condition is "terminate once the entire simulation system has completed X jobs."

1.2 THE SIMULATION TERMINATION PROBLEM

This thesis solves the simulation termination problem: given a nonterminating simulation program and any termination condition, specify how that program may be terminated at a simulation time that satisfies the termination condition. This requires accomplishing three things. The first is to find a simulation time τ when the termination condition is satisfied and all attributes needed for the termination condition calculation have been assigned their final value. The second is collect output measures at that time τ. The third is to eventually bring the simulation program to a halt.

The solution to the simulation termination problem begins in Chapter 3 with a classification of termination conditions. Termination conditions are categorized based on stability (if a condition is stable, once it becomes true it will remain true at all future times) and illustrated using space-time diagrams. Chapter 4 formalizes the simulation termination problem. It then presents a sequence of increasingly refined specifications, proof that each refined specification satisfies the problem description, and an algorithm to implement global termination conditions that are either stable or nonstable. The specifications and algorithm are architecture and simulation method independent: they apply to sequential, parallel, or distributed architectures, and to both optimistic and conservative simulation methods.

Chapter 5 addresses how the attributes needed for calculation of the termination condition and the output measures can be obtained, acknowledging that storage space is finite. Chapter 6 explores how termination conditions that are a combination of termination condition categories are implemented. Chapter 7 is a summary of how different termination conditions and combinations of termination condition categories are implemented, and chapter 8 presents the conclusion and open problems.

CHAPTER 2: REVIEW OF PREVIOUS RESEARCH

Only Lin and Lazowska [1991] have in any way addressed the simulation termination problem. Their work suggests an algorithm to terminate an optimistic parallel simulation, however they assume that each process in the simulation can independently make the decision to terminate. This assumption relates Lin and Lazowska's work to the much studied distributed termination problem, which is to determine when termination *has occurred* in a parallel program, assuming a local termination condition has caused all processes to terminate. A brief overview of this work is given in section 2.1. Another area of research which is somewhat related to this thesis is that of global state detection, an overview of this work is given in section 2.2.

2.1 DISTRIBUTED TERMINATION

2.1.1 Topic Definition

The distributed termination problem was brought to prominence by Nissim Francez [1980]. Francez noted that it may be difficult to terminate a distributed network of processors communicating via messages which are executing in cooperation. At various points in computation, an individual process may have no unprocessed input messages, thus no further actions to perform unless another message arrives. Can the process at such a point terminate? It can only terminate if it will never receive a message from another process requiring the process to perform further actions. Only when every process in the network has no more work to do and no more messages to send may termination occur. Detecting this condition given that each process is aware only of its local state is non-trivial.

2.1.2 Problem Solutions

Many algorithms have been proposed to solve the distributed termination problem. Some delegate the responsibility of termination detection to a central controller: one process or "leader" polls other processes and decides whether termination has occurred. Proposed algorithms of this type can be found in papers by Francez [1980], Dijkstra and Scholten [1980], Dijkstra et.al. [1983], Topor [1984], Apt [1986], and Rozoy [1988]. Other algorithms attempt to distribute the detection problem throughout all processors in the system, so that no central responsibility exists; this type of algorithm is preferable in networks with faults expected. Termination algorithms with no central control are presented by Cohen and Lehmann [1982], Rana [1983], Erikson [1988], and Haldar and Subramanian [1988]. Additionally, Mattern [1987] presents several decentralized algorithms based on message counting, and lists sixty-two references on the topic of distributed termination.

2.1.3 How the Simulation Termination Problem Differs From the Distributed Termination Problem

To the last, algorithms proposed to solve the distributed termination problem presume that each process can locally decide when it is able to terminate. None of these algorithms address the more general problem of detecting termination when a process needs information from other processes to make that decision. The consensus of authors mentioned in this section seems to be that: "the programmer needs only to worry about a good local condition for termination" [Cohen and Lehmann 1982]. This thesis proposes methods to accomplish termination when it is difficult or impossible to develop a good local condition for termination.

2.2 GLOBAL STATE DETECTION

2.2.1 Topic Definition

The simulation termination problem is related to the global state detection problem. The global state detection problem requires processes to record their own states and the states of communication channels so that the set of records form a global system state [Chandy and Lamport 1985] [Li et al. 1987] [Spezialetti and Kearns 1986]. These global state detection algorithms are useful for termination and deadlock detection, as well as for updating simulation clocks. Global state detection methods are also used in the Time Warp optimistic parallel simulation method [Jefferson 1985] as the basis of algorithms to calculate global virtual time (GVT) [Bellenot 1990]; GVT equals the minimum of all processes' local times and timestamps of messages in transit.

2.2.2 How the Simulation Termination Problem Differs From The Global State Detection Problem

Being able to determine a global state does not help to solve two key problems regarding implementation of a global termination condition and collection of output measures in parallel simulation. Firstly, how can output measures be retrieved from a time in the simulated past? Secondly, how can each of the asynchronous processes be brought to a halt once the termination time has been determined? These global states are useful for *local* termination detection: detecting when termination has occurred, implying a local termination condition has been used by each process to determine when it can stop. Using a global condition to *initiate* termination is not addressed in the global state detection papers mentioned above.

The most significant problem the global state detection algorithms solve is how to account for messages in transit. If the termination condition is calculated at the global virtual time when the final values of attributes are known, messages are of no concern, for they can no longer affect the values.

CHAPTER 3: CATEGORIZATION OF TERMINATION CONDITIONS

3.1 TERMINATION CONDITION DEFINITION

A Backus-Naur representation is given below to describe termination conditions examined in this thesis. Let the nonterminal <*function*> denote a mathematical function with a domain of simulation attributes (e.g., $f(t)$, where t is a simulation time). Let the nonterminal <*variable*> denote a variable (i.e., a string of letters and/or numbers that represents some value), and let <*boolean-expr*> denote a boolean expression (e.g., a < b, a = 10).

<*termination_condition*> ::= <*term*> (1)

| <*term*> ∧ <*termination_condition*> (2)

| <*term*> ∨ <*termination_condition*> (3)

<*term*> ::= <*function* > <*relation*> <*function*> (4)

| ∀<*variable*>,<*boolean-expr*> ,
<*function*><*relation*><*function*> (5)

<*relation*> ::= < | ≤ | <> | = | ≥ | >

3.2 EXAMPLES

Let the function $f(t)$ calculate the number of jobs processed by the simulation at time t. The number following each example refers to the corresponding statement (1) - (5) from the BNF expressions above:

Examples of <*term*> are:

"the number of jobs processed at simulation time t is at least X" $f(t) \geq X$ (4)

"the number of jobs processed at simulation

time t is exactly X"	$f(t) = X$	(4)
"for all simulation times less than t, the number of jobs processed is less than X"	$\forall i, t_i < t, f(t_i) < X$	(5)

Examples of <*termination_condition*> are:

"the number of jobs processed at simulation time t is at least X"	$f(t) \geq X$	(1)
"the number of jobs processed at simulation time t is exactly X"	$f(t) = X$	(2)
"the number of jobs processed at simulation time t is exactly X **and** the simulation time is at least Y"	$f(t) = X \wedge t \geq Y$	(3)
"the number of jobs processed at simulation time t is exactly X **or** the simulation time is at least Y"	$f(t) = X \vee t \geq Y$	(4)

3.3 CATEGORIES

We propose the following categories of termination conditions. A condition is stable if once it holds it will continue to hold. A condition that is not stable is called nonstable. Stable conditions are denoted by S; nonstable conditions are denoted by s.

Example (1) above represents a stable condition S, for once it holds it will continue to hold. Example (2) above represents a nonstable termination condition s. Example (3) consists of the *conjunction* of a nonstable and a stable condition ($s \wedge S$). Example (4) consists of the *disjunction* of a nonstable condition and a stable condition ($s \vee S$). These examples of termination conditions are summarized in Table 1.

Table 1: Examples of termination conditions from various termination categories

termination category	*example*
S	$f(t) \geq X$
s	$f(t) = X$
$s \wedge S$	$f(t) = X \wedge t \geq Y$
$s \vee S$	$f(t) = X \vee t \geq Y$

3.4 ILLUSTRATION USING SPACE-TIME FRAMEWORK

Chandy and Sherman's space-time diagrams [1989] will be used to represent termination conditions. A space-time diagram is a two space in which real valued simulation times are mapped to the horizontal axis, and each attribute used by a simulation program is mapped to a distinct, discrete point on the vertical axis.

Figure 1 represents a simulation with the four attributes *a1*, *a2*, *a3*, and *a4*. The times at which each attribute changes value are represented by heavy lines. For example, the value of attribute *a3* remains constant in intervals [0,4) and [4,10). Although the vertical axis of this figure is discrete, it is drawn as continuous with each attribute mapped to an interval to better illustrate the intuitive notion of the simulation program "filling in" the space-time diagram.

A stable termination condition can be illustrated by a space-time diagram such as that in Figure 2a. The termination condition is false until a certain simulation time; after that time, the termination condition is true for all subsequent times.

A nonstable termination condition can be illustrated by a space-time diagram such as that in Figure 2b or Figure 2c. Once the termination condition becomes true, it is not guaranteed to remain true for subsequent times. Some nonstable termination conditions will only be true during one interval of time: an example would be the termination condition of "exactly X jobs have been processed" (Figure 2b). Other termination conditions may hold in more than one interval of simulation time throughout the simulation. For example, in a mobile telephone simulation, "three telephone's spheres of communication overlap at simulation time t" is nonstable, because the telephones can always be in motion (Figure 2c).

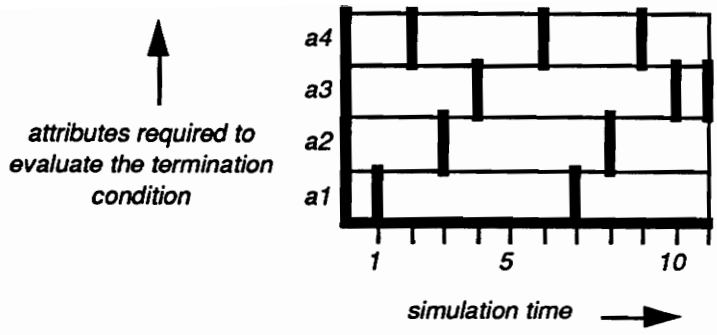


Figure 1: Space-time Diagram. This simulation consists of four attributes. Each vertical bar represents a time at which an attribute changes value. The vertical axis is discrete; the horizontal axis is continuous.

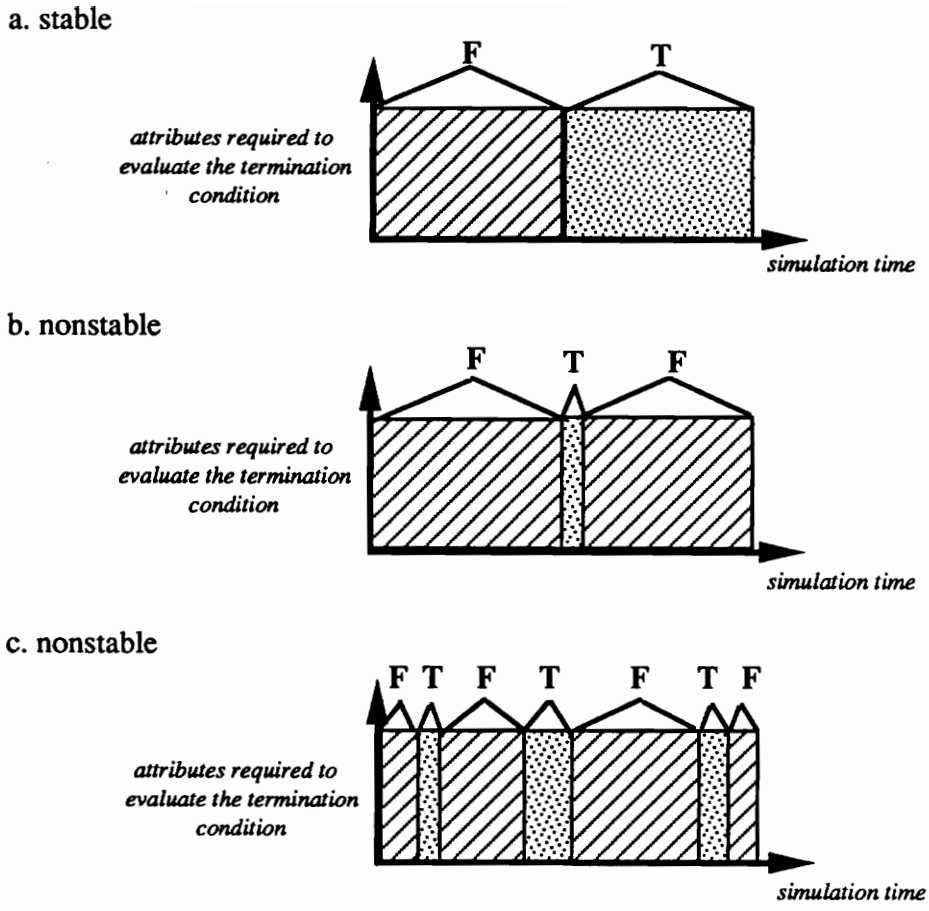


Figure 2: Stability characteristics of termination conditions. Presented using space-time diagrams.

CHAPTER 4: UNITY SPECIFICATIONS, PROOFS AND ALGORITHMS

This chapter presumes the computation model of UNITY. A description of Chandy and Misra's [1988] UNITY notation is given in the Appendix. A reader unfamiliar with UNITY should read the Appendix before proceeding.

This chapter presents specifications, proofs, and algorithms to detect nonstable and stable termination conditions occurring in simulations, report output measures at a time satisfying the termination condition, and terminate the simulation by causing the program to reach fixed point (FP). Section 4.1 presents a description of the problem. Section 4.2 presents the first specification of what must be accomplished in a simulation in order to detect a termination condition, collect output measures, and terminate the simulation by causing it to reach FP. Section 4.3 presents a refined specification that may be used to detect both stable and nonstable termination conditions. Section 4.4 refines the specification of 4.2 to yield a specification of a more efficient method to detect a stable termination condition. Section 4.5 presents an algorithm derived from the specifications. Figure 3 illustrates how these sections relate to one another.

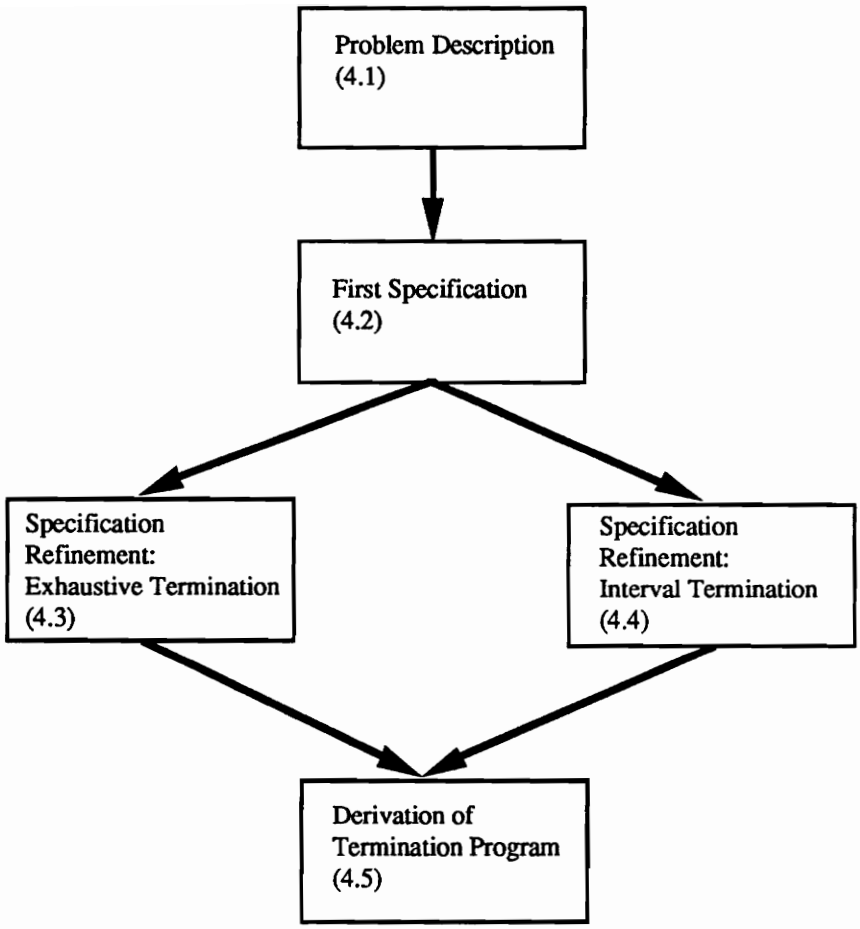


Figure 3: Specification and Refinement Diagram. This diagram maps how the refinements in chapter 4 are developed.

4.1 PROBLEM DESCRIPTION

4.1.1 Informal Problem Description

Consider any nonterminating discrete event simulation program, and a termination condition such that there exists a time t during the simulation for which the termination condition holds true and all attributes have been assigned their final values. The problem solved in this chapter is to transform the nonterminating simulation program into a simulation program that terminates at a time satisfying the termination condition.

Let τ denote a simulation time. Recalling from Chapter 3 that "time" in this thesis refers to "simulation time" (a real value), the terminating simulation program must:

(1) reach fixed point (FP),

(2) report output measure at time τ, and

(3) find a time τ such that the termination condition is satisfied and all attributes have been assigned their final values for all times less than or equal to τ.

For simplicity, we assume that in (2) above, only one output measure is calculated. Generalization to multiple output measures is straightforward, by implementing more than the one output function described in this chapter.

4.1.2 Notation Needed to Formalize Problem

This section lists and explains the notation needed to formalize the problem description. Many of the definitions below include a statement in parenthesis that illustrate properties of the term that will be developed in subsequent sections. All terms used in chapter 4 are listed in alphabetical order in Table 2 for easy reference.

attr$_i$: Let N denote a constant integer greater than or equal to one. A simulation program is represented by N attributes which are named $attr_1$, $attr_2$, ..., $attr_i$, ..., $attr_N$.

a: ($a : \Re \rightarrow \Re^N$) The domain of function *a(t)* is simulation time. Given a computation state, function *a(t)* has as its range a vector that contains the values ascribed to all $attr_i$'s at time t by the simulation program in that computation state.

af$_i$: ($af_i : \Re \rightarrow \{true,false\}$) The domain of predicate $af_i(t)$ is a simulation time. Predicate $af_i(t)$ holds if and only if the simulation program has calculated the final value of $attr_i$ at time t. Optimistic simulation methods such as Time Warp [Jefferson 1985], asynchronous relaxation [Chandy and Sherman 1989], and synchronous relaxation [Eick et. al 1991] may assign more than one value to $attr_i$ during simulation; hence we distinguish the final value assigned.

nonterm_sim: A nonterminating simulation program. Execution of *nonterm_sim* assigns values to $attr_i$.

cf: (cf: $\Re \times \Re^N \rightarrow \{true,false\}$) A function representing a boolean valued termination condition, as defined in section 3.1. The first parameter represents a candidate simulation termination time; the second parameter represents the values of $attr_i$, i = 1, 2,...N. (The termination condition is satisfied at time t if and only if $cf(t,a(t))$ is true.)

composite_prog: A program derived from *nonterm_sim* by some sequence of superpositions and unions to satisfy the formal problem statement.

τ: A program variable containing a simulation time. (A property of *composite_prog* is that at FP, $cf(\tau,a(\tau))$ and $af_i(\tau)$ hold.)

LVT_i: A program variable containing a simulation time (LVT is an acronym for "local virtual time"). (A property of LVT_i is that in a given computation state, the final value of $attr_i$ is known for all times less than or equal to LVT_i. A property of *nonterm_sim* is that the value LVT_i increases monotonically; LVT_i increases in nonuniform steps in a discrete event simulation.)

TS: *TS* denotes a sequence of times

$$TS = <<t_1,t_2,t_3,...>> \qquad \text{(definition 1a)}$$

that is nondecreasing

$$\forall\, j,t_j : 1 \le j\,,\, t_j \in TS :: t_j \le t_{j+1} \qquad \text{(definition 1b)}$$

and contains all the values of LVT_i that occur in any computation step throughout the simulation (i.e., the sequence of times at which any attribute changes value)

$$\forall\, i\, \forall j : 1 \le i \le N :: LVT_i = t[R_j] \Rightarrow t \in TS \qquad \text{(definition 1c).}$$

opf: (opf: $\Re \times \Re^N \rightarrow \Re$) A function that *composite_prog* evaluates to calculate a simulation output measure. The first parameter represents a candidate simulation termination time; the second parameter represents a set of values of $\forall i\ attr_i$ that are used in the calculation. (For example, *opf(t,a(t))* yields the value of the output measure at time t using the values ascribed to $\forall i\ attr_i$ in the current computation state.)

output_measure: A program variable assigned a value by *terminator*. (A property of *composite_prog* is that at FP, *output_measure* will contain $opf(\tau,a(\tau))$, which is the value of the output measure calculated at termination time τ.

Table 2: Summary of Notation Used in Chapter 4

name	definition
a	Given a computation state, function $a(t)$ has as its range a vector that contains the values ascribed to all $attr_i$'s at time t by the simulation program in that computation state.
af_i	Predicate $af_i(t)$ holds if and only if in the current computation state the simulation program has calculated the final value of $attr_i$ at time t.
$attr_i$	Let N denote a constant greater than or equal to one. A simulation program is represented by N attributes which are named $attr_1$, $attr_2$, ..., $attr_i$, ..., $attr_N$.
cf	(cf: $\Re \times \Re^N \rightarrow \{true, false\}$) A function representing a boolean valued termination condition, as defined in section 3.1.
composite_prog	A program derived from *nonterm_sim* by some sequence of superpositions and unions to satisfy the formal problem statement.
ETS	The initial subsequence of TS that contains all values of TS up to and including the smallest t when cf(t,a(t)) is true; the values when exhaustive termination is evaluated.
GVT	At any point in the simulation, *GVT* is less than or equal to the minimum of all LVT_i's.
ITS	The initial subsequence of *TS'* that contains all values of *TS'* up to and including the smallest t when $cf(t,a(t))$ is true; the values when interval termination is evaluated.
LVT_i	A program variable containing a simulation time. (A property of LVT_i is that in a given computation state, the final value of $attr_i$ is known for all times less than or equal to LVT_i.)
opf	(opf: $\Re \times \Re^N \rightarrow \Re$) A function that *composite_prog* evaluates to calculate a simulation output measure.
output measure	A program variable assigned a value by *terminator*. (A property of *composite_prog* is that at FP, *output_measure* will contain $opf(\tau,a(\tau))$.)
N	The number of attributes in the simulation.
nonterm sim	A nonterminating simulation program that calculates the values for $attr_i$.
term sim	A terminating simulation program.
τ	A program variable containing a simulation time. (A property of *composite_prog* is that at FP, $cf(\tau,a(\tau))$ and $af_i(\tau)$ hold.)
terminate	The boolean variable *terminate* is initially false and is set true by *terminator* once a final value is assigned to τ.
terminator	A program that cooperates with *term sim* to accomplish termination.
TS	*TS* denotes a sequence of times that is nondecreasing and contains all the values of LVT_i that occur in any computation step throughout the simulation
TS'	*TS'* denotes any sequence of times whose elements are all in *TS* and does not have an upper bound.

4.1.3 Formal Problem Description

We now have sufficient notation to formalize the problem description of 4.1.1. Assumption 1 below formalizes the assumption in section 4.1.1 that there exists a time t during the simulation for which the termination condition holds true and all attributes have been assigned their final values.

$$\exists\, t : t \in TS :: cf(t,a(t)) \wedge \langle \forall\, i : 1 \le i \le N :: af_i(t) \rangle \qquad \text{(assumption 1)}$$

The following theorem formalizes what the composite program must accomplish (i.e., (1), (2), and (3) from section 4.1.1).

Termination theorem:

Theorem 1a:

$$true \longmapsto FP \qquad \text{in } composite\_prog$$

Theorem 1b:

At FP, the output measure calculated using the final attribute values at time τ is stored in the variable *output_measure*.

$$FP \Rightarrow output\_measure = opf(\tau,a(\tau)) \qquad \text{in } composite\_prog$$

Theorem 1c:

At FP, program variable τ represents the termination time. That is, the termination condition evaluated using the final attribute values at time τ is true.

$$FP \Rightarrow cf(\tau,a(\tau)) \wedge \langle \forall i : 1 \le i \le N :: af_i(\tau) \rangle \qquad \text{in } composite\_prog$$

These theorems are proven in section 4.2.5.

4.2 FIRST SPECIFICATION

The solution strategy used is to transform *nonterm_sim* into a terminating simulation (*term_sim*) through use of a program variable named *terminate* defined below. The variable *terminate* is initially false, and program *terminator* (also defined below) sets *terminate* to true once the final value is assigned to the termination time τ. The composite program (*composite_prog*) can now be defined as the union of *term_sim* and *terminator* (definition 2 below). Figure 4 illustrates the relationship between the programs.

***term_sim*:** A terminating simulation program.

***terminate*:** A boolean program variable. (A property of the termination program is that *terminate* is initially false.)

***terminator*:** A program that cooperates with *term_sim* to accomplish termination. This program sets *terminate* to true once the final value is assigned to τ.

***composite_prog*:** The union of *term_sim* and *terminator*.

$$composite\_prog = terminator \ \square \ term\_sim \qquad \text{(definition 2)}$$

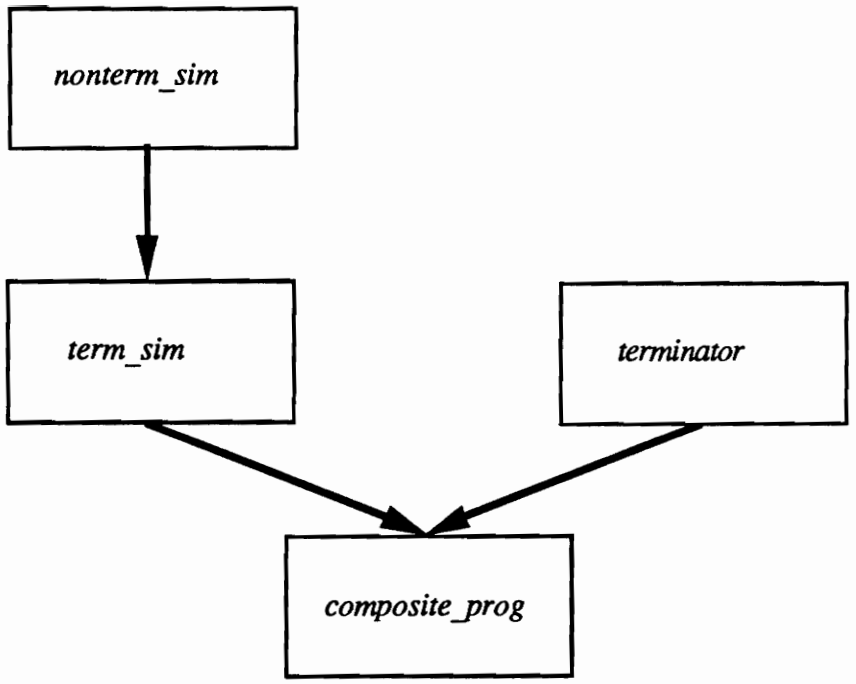


Figure 4: Development of UNITY programs. This diagram shows how the programs in chapter 4 relate to one another.

4.2.1 Specification of *nonterm_sim*

The purpose of a simulation program is to calculate the values that a set of simulation attributes attain over some interval of time. One may view *nonterm_sim* as a program that fills in the space-time rectangle that represents the simulation (as described in section 3.3). For $1 \le i \le N$, predicate $af_i(t)$ is true if and only if space coordinate i has been filled by the final value(s) of $attr_i$ for all times less than or equal to time t.

The nonterminating simulation program (*nonterm_sim*) is specified to forever make progress. "Forever make progress" is formalized by two properties: for all times less than or equal to LVT_i, all $attr_i$ values are final, and the minimum of all LVT_i values always increases.

$$< \forall\, i,t : 1 \le i \le N, t \in TS :: t \le LVT_i \Rightarrow af_i(t) > \qquad \text{in } nonterm\_sim \qquad \text{(nt 1)}$$

$$< \forall i,t : 1 \le i \le N, t \in TS :: <\min i : 1 \le i \le N :: LVT_i = t > \textbf{ ensures}$$
$$<\min i : 1 \le i \le N :: LVT_i > t >> \qquad \text{in } nonterm\_sim \qquad \text{(nt 2)}$$

4.2.2 Specification of *terminator*

This section formally states all specifications that must be fulfilled by the termination program *terminator*. A verbal description is given for each, followed by the UNITY notation for the specification.

If the *terminator* program assigns to τ a time t such that the final value of each attribute is known for that time and the termination condition evaluated using those attribute values at time t is false, then τ will be assigned the next value in the sequence TS. For simplicity, we assume that τ is initially equal to *head(TS)*.

$< \forall\ \tau,t_j : \tau,t_j \in TS :: \tau = t_j \wedge \neg cf(t_j,a(t_j)) \wedge <\forall i : 1 \leq i \leq N :: af_i(t_j) >$

ensures $\tau = t_{j+1}\ >$ in *terminator* (tm 1)

Once the variable τ is assigned a value that satisfies the termination condition, the value of τ will always satisfy the termination condition.

stable $\tau = t \wedge cf(\tau,a(\tau))$ in *terminator* (tm 2)

If *terminate* holds, then it is implied that the termination condition holds at time τ and the final attribute values are known for that time. This detects relationship also specifies that if the termination condition holds at time τ and the final attribute values are known for that time, then *terminate* will hold eventually.

terminate **detects** $cf(\tau,a(\tau)) \wedge <\forall i : 1 \leq i \leq N :: af_i(\tau) >$ in *terminator* (tm 3)

Once *terminate* is set true, the output measure is calculated using the final values of all $attr_i$ values for time τ and the result is stored in the variable *output_measure*.

terminate $\longmapsto$ *output_measure* = $opf(\tau,a(\tau))$ in *terminator* (tm 4)

Once *terminate* is set true, *terminator* has reached fixed point.

terminate $\Rightarrow FP$ in *terminator* (tm 5)

4.2.3 Derived Properties of *term_sim*

The program *term_sim* results from the superposition of the conditional "if ¬*terminate*" onto every statement in *nonterm_sim*.

program *term_sim*

transform each statement "*s*" in program *nonterm_sim* to

s if $\neg$*terminate*

transform each statement "*s* if *condition*" in program *nonterm_sim* to

s if *condition* $\wedge$ $\neg$*terminate*

end

We now derive three properties of *term_sim* from the specification of *nonterm_sim* (nt 1 and nt 2) and the above definition of *term_sim.*

Because no assignments can be made once *terminate* is set to true (by *terminator*), *terminate* implies that *FP* is reached.

$terminate \Rightarrow FP$ in *term_sim* (ts 1)

Specification nt 1 of *nonterm_sim* is identical to ts 2 given next, indicating that for all times less than or equal to LVT_i, the final $attr_i$ values have been assigned. This specification is not affected by the superposition.

$< \forall\, i,t : 1 \leq i \leq N, t \in TS :: t \leq LVT_i \Rightarrow af_i(t) >$ in *term_sim* (ts 2)

Specification nt 2 of *nonterm_sim* states that the value of LVT_i will increase. However with the superposed conditional in *term_sim*, LVT_i will only increase if $\neg$*terminate* holds. Because *terminate* will hold after *cf(t,a(t))* holds (from tm 2), progress will be made by *term_sim* while *cf(t,a(t))* is false.

$< \forall i,t : 1 \leq i \leq N, t \in TS :: <\min\, i : 1 \leq i \leq N :: LVT_i = t > \wedge \neg cf(t,a(t))$

ensures $<\min\, i : 1 \leq i \leq N :: LVT_i > t >>$ in *term_sim* (ts 3)

4.2.4 Derived Properties of *composite_prog*

Earlier in section 4.2, *composite_prog* was defined as the union of *term_sim* and *terminator*. Generally, if a specification holds for a component program, it holds for the composite program, for the union operation merely merges the assignment statements from the component programs. However, it must be assured that no conflicts (variables written to by both programs) exist between the two sets of specifications.

The only variables that are shared between *term_sim* and *terminator* are *terminate* and *a(t)*. The variable *terminate* is written to by *terminator* (it is set to true when *cf(t,a(t))* holds) and is only read by *term_sim*. The vector *a(t)* is written to by *term_sim* and is only read by *terminator*. This is formalized as follows:

constant *terminate*	in *term_sim*
constant $< \forall t : t \in \Re :: a(t) >$	in *terminator*.

Therefore, tm 1, tm 2, tm 3, and tm 4 from *terminator* and ts 2 and ts 3 from *term_sim* can be restated as specifications that hold for *composite_prog*, for no conflicts exist between them (renamed cp 1, cp 2, cp 3, cp 4, cp 5, and cp 6 respectively). The specifications tm 4 and ts 1 are identical, thus can be restated as cp 7 below.

Specification 1:

A summary of the UNITY specifications for *composite_prog* are given below.

$< \forall\ \tau,t_j : \tau,t_j \in TS :: \tau = t_j \wedge \neg cf(t_j,a(t_j)) \wedge <\forall i : 1 \le i \le N :: af_i(t_j) >$

ensures $\tau = t_{j+1}$ $>$ in *composite_prog* (cp 1)

stable $\tau = t \wedge cf(\tau,a(\tau))$ in *composite_prog* (cp 2)

terminate **detects** $cf(\tau,a(\tau)) \wedge <\forall i : 1 \leq i \leq N :: af_i(\tau) >$

in *composite_prog* (cp 3)

terminate $\longmapsto$ *output_measure* = $opf(\tau,a(\tau))$ in *composite_prog* (cp 4)

$< \forall i,t : 1 \leq i \leq N, t \in TS :: t \leq LVT_i \Rightarrow af_i(t) >$ in *composite_prog* (cp 5)

$< \forall i,t : 1 \leq i \leq N, t \in TS :: <\min i : 1 \leq i \leq N :: LVT_i = t > \wedge \neg cf(t,a(t))$

ensures $<\min i : 1 \leq i \leq N :: LVT_i > t >>$ in *composite_prog* (cp 6)

terminate $\Rightarrow$ *FP* in *composite_prog* (cp 7)

4.2.5 Proof that the Specification 1 Implies the Problem Description

This section shows that the specification 1 implies the formal problem description given in 4.1.3. To reiterate, Theorem 1a states that *composite_prog* reaches fixed point. Theorem 1b states that *composite_prog* reports the output measures at time τ, and Theorem 1c states that *composite_prog* assigns to τ a time t such that the value of each attribute is known for that time, and the termination condition evaluated using the final attribute values at time t is true.

Proof of Theorem 1a:

(1) *true* $\longmapsto$ *terminate* ,from (assumption 1) and (cp 3)

(2) *terminate* $\Rightarrow$ *FP* , (cp 7)

(3) *true* $\longmapsto$ *FP* , transitivity on (1) and (2)

Proof of Theorem 1b:

(1) *terminate* $\longmapsto$ *output_measure* = $opf(\tau,a(\tau))$

, (cp 4).

(2) *terminate* $\Rightarrow$ *FP* , (cp 7)

(3) FP $\Rightarrow$ *output_measure* = $opf(\tau, a(\tau))$

, follows (1) and (2)

Proof of Theorem 1c:

(1) $< \exists\, t : t \in TS :: cf(t.a(t)) \wedge < \forall\, i : 1 \leq i \leq N :: af_i(t) > >$

, (assumption 1)

(2) $< \forall\, \tau, t_j : \tau, t_j \in TS :: \tau = t_j \wedge \neg cf(t_j, a(t_j)) \wedge < \forall i : 1 \leq i \leq N :: af_i(t_j) >$

ensures $\tau = t_{j+1}$ > , (cp 1)

(3) **stable** $\tau = t \wedge cf(\tau, a(\tau))$, (cp 2)

(4) *terminate* $\Rightarrow$ FP , (cp 7)

(5) *terminate* **detects** $cf(\tau, a(\tau)) \wedge < \forall i : 1 \leq i \leq N :: af_i(\tau) >$

, (cp 3)

(6) FP $\Rightarrow$ $cf(\tau, a(\tau)) \wedge < \forall i : 1 \leq i \leq N :: af_i(\tau) >$

, follows from (2) - (5) above

4.3 SPECIFICATION REFINEMENT: EXHAUSTIVE TERMINATION

Studying specification 1, it is clear that if the final value of each attribute at some time *t* is known, and if the termination condition evaluated using the final attribute values at time *t* is true, then a single statement in *composite_prog* can cause *terminate* to become *true*. This implies FP by cp 7 and output measure calculation by cp 4, thus satisfying two of the three theorems comprising the simulation termination problem formalized in 4.1. It is assumed that such a time *t* as mentioned above does exist (assumption 1), so the termination time τ will be assigned that time (cp 1), and the Theorem 1c from 4.1.3 (FP $\Rightarrow$

$cf(\tau,a(\tau))$ $\wedge$ $<\forall i : 1 \leq i \leq N :: af_i(\tau) >$) will be satisfied. What is not obvious is how that time t is found.

The exhaustive termination method proposed below exhaustively tests every $t \in TS$ in increasing order until a t that can be assigned to τ is found. Exhaustive termination will detect both stable and nonstable termination conditions.

4.3.1 Introduction of GVT

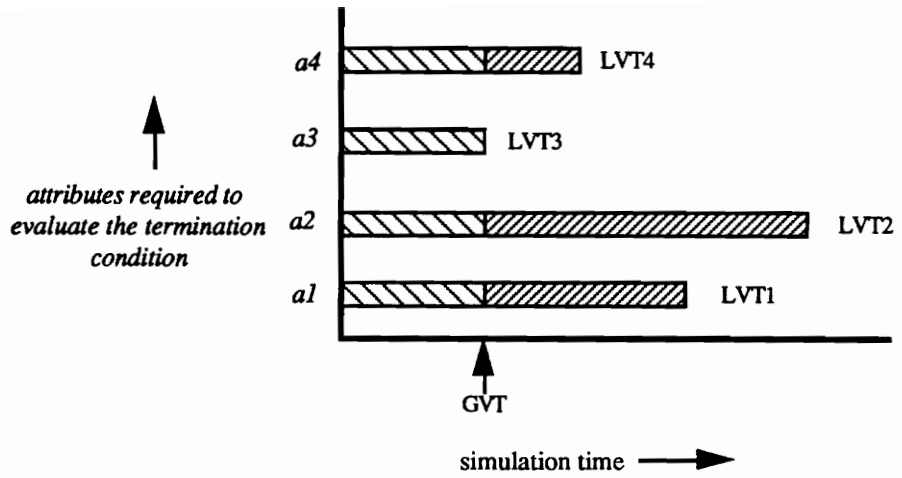
To specify exhaustive termination, we will introduce the program variable *GVT* (using Jefferson's [1985] acronym referring to "global virtual time"). At any point in the simulation, *GVT* is less than or equal to the minimum of all current LVT_i's.

$$GVT \leq < \min i : 1 \leq i \leq N :: LVTi > \qquad \text{(definition 3)}$$

4.3.2 Informal Description of the Solution

We want to find a computation state in which the value of *GVT* satisfies *cf(GVT,a(GVT))*.

It is possible for each $attr_i$ in the simulation to have a different LVT_i at any given computation step if the simulation program is mapped to an asynchronous parallel computer. Thus, the *cf(t,a(t))* function can only be calculated at times t that are less than or equal to *GVT*, implying that all $attr_i$'s are the final values for that time t. In other words, a property of *GVT* is that all attribute values have been assigned their final values for times less than or equal to *GVT*; this property can be derived from combining definition 3 and cp 5. See Figure 5.



KEY:

: times that may be used to calculate the termination condition ($af_i(t)$ holds)

: times that may not be used to calculate the termination condition ($af_i(t)$ does not hold)

Figure 5: Illustration of *GVT* in Relation to LVT_i. Each $attr_i$ contributing to the simulation may have a different simulation time (LVT_i). To ensure final attribute values have been assigned to calculate the termination condition $cf(t,a(t))$, the time t used may not be larger than *GVT*.

4.3.3 Definition of Sequence *ETS*

An additional definition is used to simplify the specification. *ETS* is defined as the initial subsequence of *TS* that contains all values of *TS* up to and including the smallest t such that *cf(t,a(t))* is true. Because of definition 1c and assumption 1, the sequence *TS* is guaranteed to contain a time t that satisfies *cf(t,a(t))*.

$ETS \sqsubseteq TS$ (definition 4a)

$cf(last(ETS),a(last(ETS)))$ (definition 4b)

$< \forall\, t : t \in ETS \wedge t \neq last(ETS) \;::\; \neg cf(t,a(t)) >$ (definition 4c)

4.3.4 Formal Description of the Solution Strategy

Exhaustive termination is specified by cp 2, cp 3, cp 4, cp 5, cp 6, and cp 7 from specification 1 in 4.2.4, with cp 1 being replaced by cp 8 and cp 9 specified below.

Specification 2: [(cp 2 - cp 7) plus (cp 8 - cp 9) given next.]

GVT will assume values in *ETS* in increasing order until one that satisfies the termination condition is reached. Initially, *GVT* will hold the value of *head(ETS)* (this is included in the initially section of the program in figure 8).

$< \forall\, j,t_j : 1 \leq j\,,\, t_j \in ETS :: GVT = t_j \wedge \neg cf(t_j,a(t_j)) \wedge <\forall i : 1 \leq i \leq N :: af_i(t_j) >$

ensures $GVT = t_{j+1}\ >$ (cp 8)

By cp 8 and definition 4b, *GVT* will eventually equal the time when the termination condition holds true. Once *GVT* equals *last(ETS)*, then the termination time τ is assigned the value of *GVT* and *terminate* is set to true.

$GVT = last(ETS)$ **ensures** $\tau = GVT \wedge terminate$ (cp 9)

4.3.5 Proof of Correctness of the Solution Strategy

We wish to show that if specification 2 is met, then cp 1 follows (i.e., FP implies a suitable termination time τ has been found). Therefore Theorem 1c, which was proven using cp 1 in 4.2.5, can also be proven by the substitution made in specification 2.

(1) $GVT \leq < \min i : 1 \leq i \leq N :: LVTi >$, (definition 3)

(2) $< \forall\, i,t : 1 \leq i \leq N, t \in TS :: t \leq LVT_i \Rightarrow af_i(t) >$, (cp 5)

(3) $\forall i : 1 \leq i \leq N :: af_i(GVT)$, follows from (1) and (2) above

(4) $true \longmapsto GVT = last(ETS)$, by (cp 8) and (definition 4c)

(5) $cf(last(ETS),a(last(ETS))$, (definition 4b)

(6) $true \longmapsto cf(GVT, a(GVT))$, by (4) and (5) above

(7) $true \longmapsto cf(GVT, a(GVT)) \wedge <\forall i : 1 \leq i \leq N :: af_i(GVT) >$, by (3) and (6) above

(8) $GVT = last(ETS)$ **ensures** $\tau = GVT \wedge terminate$, (cp 9)

(9) $true \longmapsto terminate$, (4) and (8)

(10) $terminate \Rightarrow FP$, (cp 7)

(11) $FP \Rightarrow cf(GVT,a(GVT)) \wedge <\forall i : 1 \leq i \leq N :: af_i(GVT) >$, by (7), (9), and (3)

(12) $FP \Rightarrow cf(\tau,a(\tau)) \wedge <\forall i : 1 \leq i \leq N :: af_i(\tau) >$, Theorem 1c, equivalent to (11) with $\tau = GVT$.

4.3.6 Motivation for the Next Refinement: Efficiency of *composite_prog*

Testing all times in *TS* to guarantee that we find a time *t* satisfying *cf(t,a(t))* is not necessary if the termination condition is stable (meaning that once it becomes true it will remain true). The stability permits the termination condition to be calculated in intervals (i.e., every so often) rather than exhaustively. This idea is illustrated in Figure 6. Knowledge that once the condition becomes true it will remain true ensures that this method will be able to find a $t \in TS$ that satisfies *cf(t,a(t))*.

Being able to use an interval termination scheme rather than an exhaustive one should cause the overall wall clock time required by the simulation to be reduced in most cases. The reasoning behind this statement can be seen with the time lines of Figure 7. The vertical lines on each horizontal time line indicate individual assignment statements performed by the simulation. Line (b) indicates the timing of the simulation before a termination method is incorporated. Line (a) illustrates the modified timing when the termination condition is exhaustively calculated after every assignment statement. Each one will take longer; the termination condition will occur at a later wall clock time. Line (c) illustrates the timings when (b) is modified to include interval termination. Even though termination will not be detected *as soon as* the termination condition becomes true, it should be the case that the simulation will complete quicker than if the exhaustive method is used due to smaller overhead of exhaustive calculations.

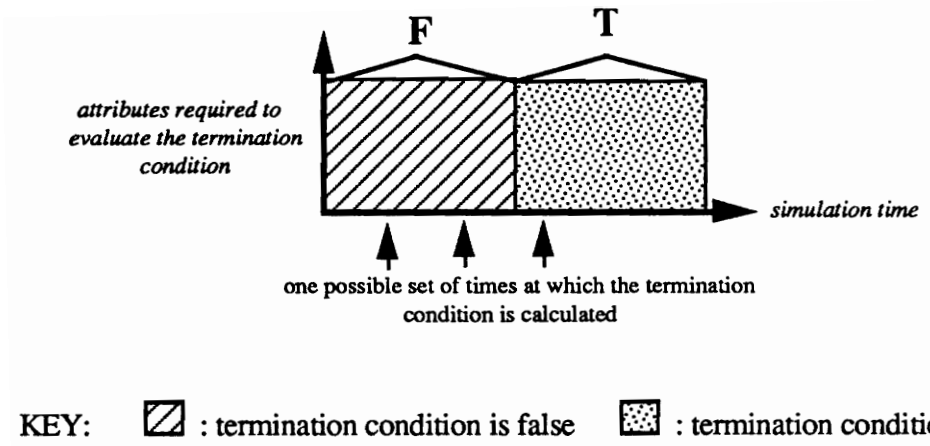
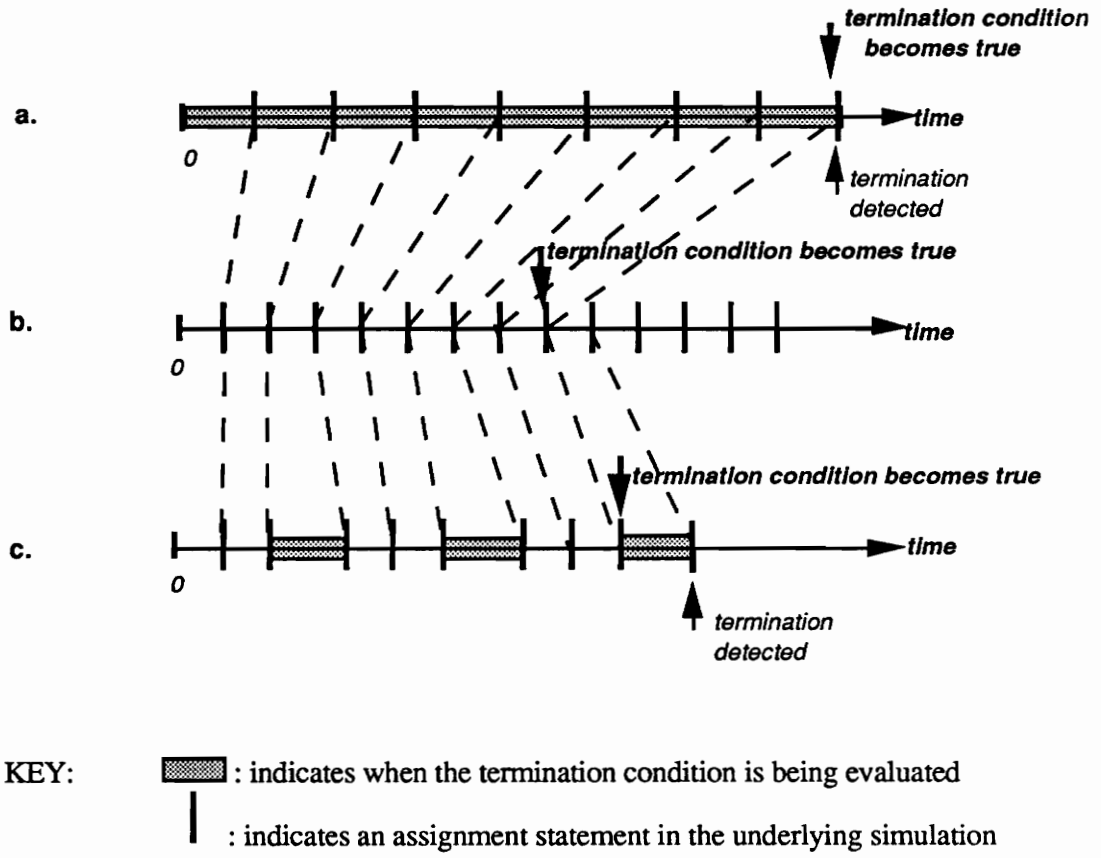


Figure 6: Interval Evaluation of a Stable Termination Condition. Knowledge that once the condition becomes true it will remain true ensures that this method will be able to find a time that satisfies the condition.



KEY: : indicates when the termination condition is being evaluated

: indicates an assignment statement in the underlying simulation

Figure 7: Justification of the Interval Termination Method. Time line (b) represents the underlying simulation. The comparison between lines (a) and (c) shows that if the termination condition is evaluated exhaustively as in (a), termination will likely be detected at a later time than if it is evaluated over intervals as in (c).

4.4 SPECIFICATION REFINEMENT: INTERVAL TERMINATION

Interval termination can be used to implement a stable termination condition. This method evaluates the termination condition over intervals rather than exhaustively evaluating the termination condition whenever an attribute changes.

4.4.1 Determining Interval Length - a Question of Efficiency

How often should the termination condition be calculated during a simulation for maximum efficiency? Some penalty of speed (and/or storage) is likely to occur for every calculation of the termination condition in determining the appropriate attributes needed by the termination detector and saving the attributes until the condition can be calculated. Thus is the argument for making the interval between calculations as large as possible. The counter argument is that if intervals are large, the fact that the termination condition has become true may not be detected until quite a while after it happens, thus termination detection is delayed.

The interval length to allow maximum efficiency is problem dependent. If it is known before execution of a simulation that the simulation will require at least five hours to complete, a larger calculation interval would be used than if the simulation was thought to complete in minutes. Characteristics of the problem can be used for the best guess, however the exact answer could not be known unless the termination time were known beforehand, then termination detection would not be an issue!

4.4.2 Definition of Stable Termination Function

If the termination function is stable, once it becomes true it will remain true at all future times. This is formalized in UNITY below.

stable termination function =

$$<\forall t,t' : t,t' \in TS,\ t' > t :: cf(t,a(t)) \Rightarrow cf(t',a(t'))> \qquad \text{(definition 5)}$$

If the termination condition is evaluated at any time greater than the time when the termination condition became true, that time will satisfy the termination condition.

4.4.3 Definition of Sequence *ITS*

The sequence *TS* was defined in 4.2 as the sequence of times at which attributes are assigned new values. Let *TS'* denote any time ordered sequence of unbounded length whose elements are all in *TS*.

$$TS' = <<t_1,t_2,...>> \qquad \text{(definition 6a)}$$

$$t \in TS' \Rightarrow t \in TS \qquad \text{(definition 6b)}$$

$$\forall j,t_j : 1 \leq j,\ t_j \in TS' :: t_j \leq t_{j+1} \qquad \text{(definition 6c)}$$

Let *ITS* denote the initial subsequence of *TS'* up to and including the smallest time in *TS'* when *cf(t,a(t))* is true. It follows from definition 5 (*cf(t,a(t))* is stable) and definition 6a (*TS'* is unbounded) that the sequence *TS'* contains a time that satisfies *cf(t,a(t))*.

$$ITS \sqsubseteq TS' \qquad \text{(definition 7a)}$$

$$cf(last(ITS),a(last(ITS))) \qquad \text{(definition 7b)}$$

$$<\forall\, t : t \in ITS \wedge t \neq last(ITS) :: \neg cf(t,a(t))> \qquad \text{(definition 7c)}$$

4.4.4 Formal Description of the Solution Strategy

Interval termination is described by cp 2, cp 3, cp 4, cp 5, cp 6, and cp 7 from specification 1 in 4.2.4, with cp 1 being replaced by cp 10 and cp 11 specified below. This specification will detect stable termination conditions.

Specification 3 [**stable** termination function, (cp 2 - cp 7) plus (cp 10 - cp 11) given next.]

GVT will assume values in *ITS* in increasing order until one that satisfies the termination condition is reached. Initially, *GVT* will equal *head(ITS)* (this is included in the initially section of the program in figure 8).

$$< \forall\, j,t_j : 1 \le j\,,\, t_j \in ITS :: GVT = t_j \wedge \neg cf(t_j,a(t_j)) \wedge <\forall i :1\le i \le N:: af_i(t_j) > \textbf{ ensures } GVT = t_{j+1} > \qquad \text{(cp 10)}$$

By cp 10 and definition 7b, *GVT* will eventually equal the time when the termination condition holds true. Once *GVT* equals *last(ITS)*, then the termination time τ is assigned the value of *GVT* and *terminate* is set to true.

$$GVT = last(ITS) \textbf{ ensures } \tau = GVT \wedge terminate \qquad \text{(cp 11)}$$

4.4.5 Proof of Correctness of the Solution Strategy

We wish to show that if specification 3 is met, then cp 1 follows (i.e., FP implies a suitable termination time τ has been found). Therefore theorem 1c, which was proven using cp 1 in 4.2.5, can also be proven by the substitution made in specification 3.

(1) $< \min i : 1 \le i \le N :: GVT \le LVTi >$, (definition 3)

(2) $< \forall\, i,t : 1 \le i \le N, t \in TS :: t \le LVT_i \Rightarrow af_i(t) >$

, (cp 5)

(3) $\forall i : 1 \leq i \leq N :: af_i(GVT)$, follows from (1) and (2) above

(4) $true \longmapsto GVT = last(ITS)$, by (cp 10) and (definition 5c)

(5) $cf(last(ITS),a(last(ITS))$, (definition 5b)

(6) $true \longmapsto cf(GVT, a(GVT))$, by (4) and (5) above

(7) $true \longmapsto cf(GVT, a(GVT)) \wedge <\forall i : 1 \leq i \leq N :: af_i(GVT)>$

, by (3) and (6) above

(8) $GVT = last(ITS)$ **ensures** $\tau = GVT \wedge terminate$

, (cp 11)

(9) $true \longmapsto terminate$, (4) and (8)

(10) $terminate \Rightarrow FP$, (cp 7)

(11) $FP \Rightarrow cf(GVT,a(GVT)) \wedge <\forall i : 1 \leq i \leq N :: af_i(GVT)>$

, by (7), (9), and (3)

(12) $FP \Rightarrow cf(\tau,a(\tau)) \wedge <\forall i : 1 \leq i \leq N :: af_i(\tau)>$

, Theorem 1c, equivalent to (11) with $\tau = GVT$.

4.5 DERIVATION OF A PROGRAM FROM THE SOLUTION STRATEGY SPECIFICATIONS

4.5.1 Shared Variables Between *term_sim* and *terminator*

As discussed in section 4.2.4, two constructs are shared between the simulation program and the termination program: *terminate* and *a(t)*. The variable *terminate* is straightforward to represent in a programming notation: it is a boolean. Implementation of function *a(t)* will be considered in chapter 5.

```
declare
   a : array[1..M] of attr_vector      {Storage array for attr_i values.
                                        The structure "attr_vector" contains
                                        N reals to hold attr_i values.
                                        M denotes the time index.
                                        Sim_progwill write to this array,
                                        terminator will read. }
   terminate: boolean                  {Once this is set true by terminator,
                                        term_sim will reach FP. }
```

4.5.2 *GVT* Updates: The Difference Between Exhaustive and Interval Termination

The variable *GVT* is updated by an external program, and *GVT* updates are presented to *terminator* as the sequence *GVTS*. The intricacies of the program that calculates *GVT* are not explored here, for many *GVT* algorithms exist in the literature (see Bellenot [1990]).

```
GVTS : sequence of real          {Indicates the updated values of GVT.}
```

The difference between exhaustive termination and interval termination algorithms will be in the frequency of *GVT* updates. To implement exhaustive termination, the *GVT* value must be precisely known for every time throughout the simulation; *GVTS* will equal *ETS*. To implement interval termination, *GVT* only needs to be updated periodically; this is how the *GVT* algorithm is traditionally executed. *GVTS* will equal *ITS*. Clearly, the continual *GVT* calculation required by exhaustive termination could represent a significant efficiency penalty.

4.5.3 Termination Algorithm

Figure 8 presents a program satisfying specification 2 (exhaustive termination) or specification 3 (interval termination) presented in this chapter. As mentioned above, the

difference between implementing the two specifications is only in how often *GVT* is updated.

Evaluation of functions *cf(t,a(t))* and *opf(t, a(t))* in a parallel simulation typically requires an algorithm to efficiently combine attribute values that may be private to two or more processes. We propose using Lakshman and Wei's [1988] algorithm to evaluate these functions.

```
Program terminator

    always

        GVT = head(GVTS)

    initially

        terminate = false

    assign

    {Evaluate the termination function to see if termination can occur.}

        terminate              , τ        , GVTS       :=
        cf(GVT, a(GVT))        , GVT      , tail(GVTS)      if   ¬ terminate

    [] {If termination can occur, calculate output for at the termination time.}

        output_measure := opf(τ,a(τ))                       if   terminate

end
```

Figure 8: Solution to the Simulation Termination Problem

CHAPTER 5: OBTAINING ATTRIBUTE VALUES

Only one structure in Figure 8 can not be readily implemented in a programming language: *a(t)*, the vector that contains the *N* final values of $attr_i$ at time *t*. This chapter describes how to implement *a(t)*.

It must be noted that the vector *a(t)* contains the attributes needed for calculation of the termination condition (denoted *tca* for termination condition attributes) and the attributes needed for calculation of the output measures (denoted *oma* for output measure attributes). The attributes of *tca* and *oma* may be identical, may overlap, or may be disjunct. Consider two examples of simulation programs. In the first example, the termination condition is "the total number of jobs processed is at least *X*" and the output measure desired is the average processing time of jobs. In this case, *tca* and *oma* are identical. In the second example, the termination condition is the same, but the output measure desired is some function of any measurable quantity produced per job during the simulation. In this case, *tca* and *oma* are different.

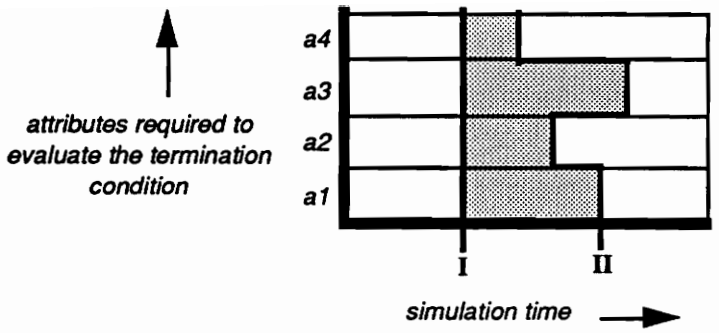
Section 5.1 addresses the collection of *tca*. Section 5.2 addresses the collection of *oma*. Note that if *tca* and *oma* are identical, the methods given in 5.2 will not be necessary, because all attributes needed for the output measures will be the same as the attribute values needed for the termination condition.

5.1 OBTAINING ATTRIBUTE VALUES NEEDED FOR CALCULATION OF THE TERMINATION CONDITION

Execution of the nonterminating simulation program (*nonterm_sim* from chapter 4) will determine the values of the attributes needed for calculation of the termination

condition. Calculated attributes must be available to the termination program (*terminator*) until it is determined that the termination condition is not satisfied at that time t, after which the values that correspond to that time may be discarded. The problem solved here is to ensure availability of the needed attribute values, while acknowledging that storage space in a computer is finite. It may not be assumed that every value of every attribute can be stored for a simulation of any significant size.

Figure 9 is a space-time diagram to illustrate the portion of the simulation where attribute values must be available. Line (I) indicates the earliest time t for which the termination condition has not yet been calculated. This time t equals the time held by the variable *GVT*, for *GVT* does not increase until the termination condition is calculated using the attribute values corresponding to *GVT* (see Figure 8). Line (II) indicates the latest time t for which the final value of each attribute is known (LVT_i). All attribute values that correspond to times to the right of line (I) (i.e., greater than *GVT*) must be available for future termination condition calculations.



I: Indicates the earliest time t for which the termination condition has not yet been calculated (GVT).

II: Indicates the latest time t for which the final value of each attribute is known (LVT_i).

 : Indicates the times for which the attribute values must be stored.

Figure 9: Needed Attribute Values for Termination Condition Calculation. This space-time diagram indicates the execution of a simulation program. All attribute values that correspond to times to the right of line (**I**) must be available for future termination condition calculations.

5.2 OBTAINING ATTRIBUTE VALUES NEEDED FOR CALCULATION OF OUTPUT MEASURES

This section explains three strategies for collection of attribute values needed for calculation of output measures that were proposed by Abrams and Richardson [1991]: dissociative, retrospective, and prospective.

5.2.1 The Dissociative Strategy

This strategy requires that all attribute values needed for the output measures (*oma*) are stored, just as the attribute values needed for the termination condition (*tca*) are. Execution of *nonterm_sim* will determine the values of the attributes needed for calculation of the output measures. Calculated attributes must be available to the termination program (*terminator*) until it is determined that the termination condition is not satisfied at that time *t*, after which the values that correspond to that time may be discarded.Thus, at all points in the simulation, the termination program has all the information it needs to evaluate the termination condition and calculate output measures.

Figure 10a shows the portion of the space-time diagram corresponding to the attribute values that must be available to the termination program. As with termination condition calculation, attribute values corresponding to times less than *GVT* may be discarded.

5.2.2 The Retrospective Strategy

This strategy offers a method of obtaining the needed attributes without storing every *oma* value at every time greater than *GVT*. The retrospective strategy requires only that the *oma* values are known for some time *t* that is less than or equal to the termination

time τ. Once the termination time has been determined, simulation execution "rolls back" or returns to the time t in the simulated past for which the *oma* values are known, and proceeds to calculate the *oma* values up to the termination time.

Returning simulation execution to a time in the simulated past is possible when using the Time Warp [Jefferson 1985] parallel simulation method, because state saving is an inherent part of the method. This strategy could not be implemented without significant changes to a conservative parallel simulation protocol such as BCM [Bryant 1977; Chandy et al. 1979] because state saving is not a part of the standard algorithm.

Figure 10b shades the portion of the space-time diagram corresponding to the *oma* values that must be available to the termination program for retrospective termination.

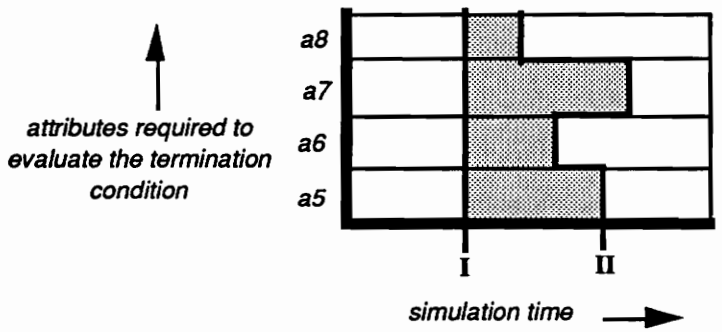
5.2.3 The Prospective Strategy

This method proposes acquiring the needed *oma* values from a time in the simulated *future*. When the termination program finds a termination time τ, it then determines and informs the simulation program of a time in the simulated future when simulation program execution can terminate. Therefore, the *oma* value only needs to be known for the current LVT_i. At τ, all LVT_i's will greater than or equal to τ.

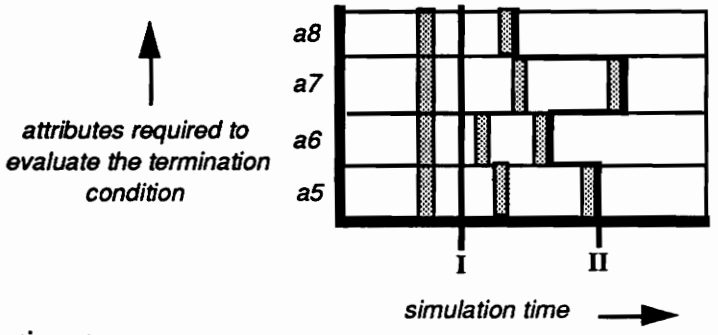
Prospective termination can only be used if the termination condition for the simulation is stable, implying that once the condition holds true it will hold true at all future times. If the termination condition is not stable, it could not be ensured that the condition holds at any time in the simulation future.

Figure 10c shades the portion of the space-time diagram that corresponds to the *oma* values that must be available to the termination program for prospective termination.

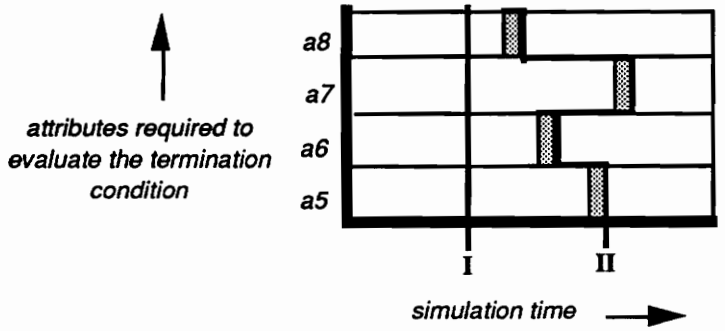
a. dissociative strategy



b. retrospective strategy



c. prospective strategy



I : The earliest time t for which the termination condition has not yet been calculated (GVT).

II : The latest time t for which the final value of each attribute is known (LVT_i).

: Indicates the times for which the attribute values must be stored.

Figure 10: Needed Attribute Values for Output Measure Calculation. Attribute values that corresponding to the shaded regions must be available for possible output measure calculations. The regions indicated in (b) were arbitrarily chosen; any regions are acceptable provided that there is at least one shaded region to the left of line I, and the horizon is shaded for each attribute.

CHAPTER 6: COMBINATORIAL TERMINATION CONDITIONS

As was described in section 4.1, a termination condition can be a combination of other termination conditions. This section explores the implications of such combinations, specifically how the two methods presented in chapter 4 of implementing a termination condition, interval and exhaustive, can be used to implement the combinations.

6.1 CONJUNCTIVE TERMINATION CONDITIONS

In section 4.1, it was noted that a termination condition could consist of a conjunction of terms. For example, the termination condition that a simulation programmer may wish to use is "the number of jobs processed at simulation time *t* is at least N **and** the simulation time is at least t". Another example could be "the number of jobs processed at simulation time *t* is between N and M". This could be broken down into the conjunct of two conditions: #jobs > N $\wedge$ #jobs < M.

Solutions to solve the three combinations of types of terms must be described: the conjunction of more than one stable condition, the conjunction of more than one nonstable condition, and the conjunction of stable and nonstable conditions.

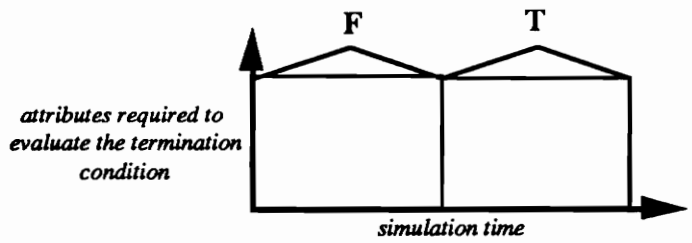
The first two combinations are straightforward to solve. Because a stable condition can be solved using the interval termination method, so can the conjunction of stable conditions. Once the stable conditions become true they will remain true: thus the conjunction of stable conditions is a stable condition.

The conjunction of nonstable conditions is probably a nonstable condition, thus the exhaustive method is required to solve the condition. However it is possible that such a conjunction can be stable, thus the interval method may be used. For example, suppose

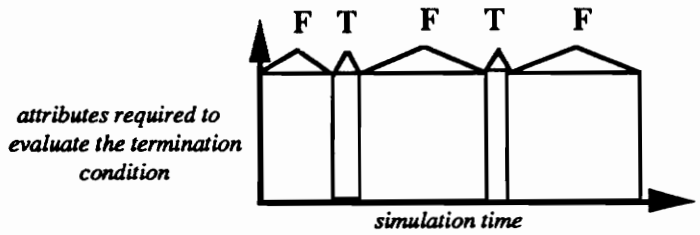
one nonstable condition is of the form "true for times t_1 through t_3, false for times t_4 and t_5, and true for all times greater than t_5", and a second nonstable condition is of the form "false for times t_1 and t_2, true for times t_3 and t_4, false for time t_5, and true for all times greater than t_5". The conjunction of these two nonstable conditions is a stable condition: false up to and including time t_5, and true for all times greater than t_5.

Figure 11 illustrates such a combination of stable and nonstable termination conditions. Figure 11a illustrates a stable property, 11b a nonstable property, and 11c represents the conjunction of the two. The overall termination condition is satisfied and termination can occur only when both properties are concurrently true.

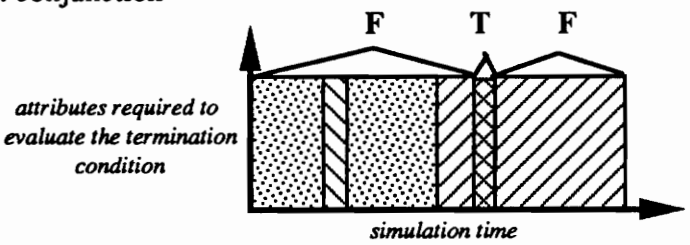
a. stable



b. nonstable



c. conjunction



KEY for c:

: both conditions are false

: nonstable true, stable false

: stable true, nonstable false

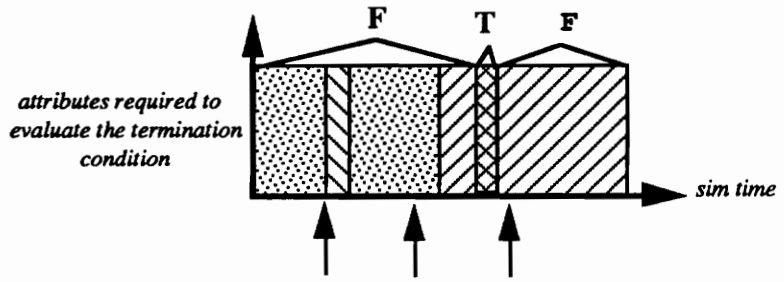
: both conditions are true

Figure 11: The Conjunction of a Stable and a Nonstable Condition. (c) is an illustration of the conjunction between a stable condition (a), a nonstable condition (b). Only when both conditions are true simultaneously can termination occur.

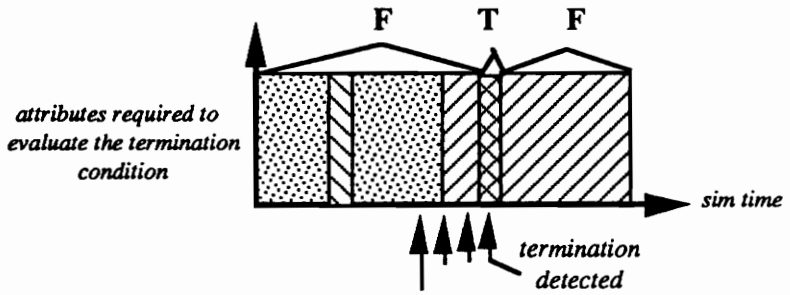
The conjunction of stable and nonstable conditions allows for the possibility of a termination method that combines both previously described termination methods. Such a combination could be solved using the exhaustive method; however, because it is known that the stable condition is initially false, and remains false until it becomes true (by the definition of **stable**), any exhaustive calculation before the stable condition becomes true will be unproductive.

The proposed scheme to solve the conjunction of stable and nonstable conditions is as follows: use the interval termination method until the first time is found where the stable condition(s) is(are) true, then backup (restore the attributes) to the last false point, and begin searching exhaustively for a time that will satisfy the combination termination condition. The "Time Warp" optimistic parallel simulation method [Jefferson 1985] would lend itself well to implementing this, because of the "rollback" mechanism inherent in the method. The "interval then exhaustive" termination method is illustrated in Figure 12, continuing the example from Figure 11.

First phase : interval search



Second phase: exhaustive search



KEY :

: both conditions are false

: nonstable true, stable false

: stable true, nonstable false

: both conditions are true

: indicates where termination condition is evaluated

Figure 12: Interval_then_Exhaustive Method to Detect Termination. When the termination condition is a conjunction of stable and nonstable conditions. In the first phase, the interval termination method is used, until the first test where the stable condition is true is found. The second phase backtracts to the last interval calculation (where the stable condition was false), and begins an exhaustive search to find where both conditions become true.

6.2 DISJUNCTIVE TERMINATION CONDITIONS

In section 4.1, it was noticed that a termination condition could consist of a disjunct of terms. For example, the termination condition that a simulation programmer may wish to use is "the number of jobs processed is at least N **or** the simulation time is at least X".

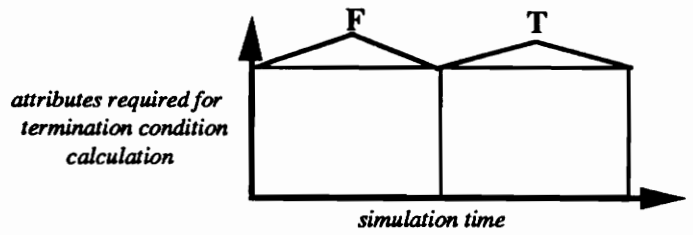
Solutions to solve the three combinations of types of terms must be described: the disjunct of stable conditions, the disjunct of nonstable conditions, and the disjunct of stable and nonstable conditions.

The first two combinations are straightforward to solve. Because a stable condition can be solved using the interval termination method, so can the disjunct of stable conditions. Once the stable conditions become true they will remain true: thus the disjunct of stable conditions is a stable condition.

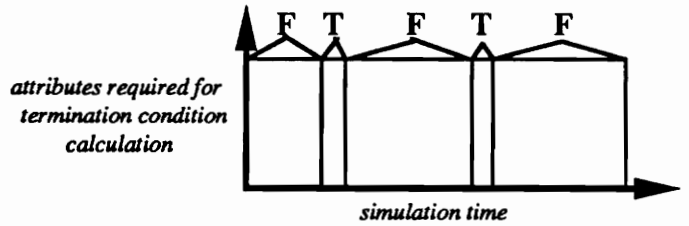
The disjunction of nonstable conditions is probably a nonstable condition, thus the exhaustive method is required to solve the condition. However as mentioned in 6.1, it is possible that such a disjunction can be stable, thus the interval method may be used.

The disjunct between stable and nonstable termination conditions does not have a straightforward solution strategy. Figure 13 illustrates this possibility. Figure 13a represents a stable condition, 13b a nonstable condition, and 13c represents the disjunct of the two conditions. Whenever at least one of the conditions is true termination may occur.

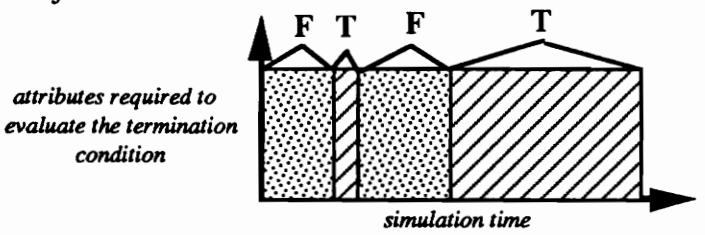
a. stable



b. nonstable



c. disjunct



KEY for c:

: both conditions are false

: at least one of the conditions is true

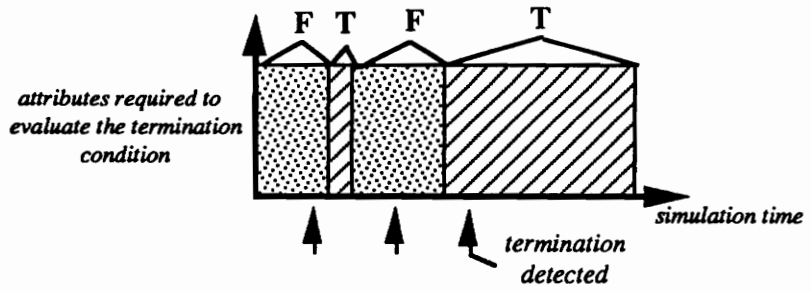
Figure 13: The Disjunct of a Stable and a Nonstable Condition. (c) is an illustration of the disjunct between a stable condition (a), and a nonstable condition (b). Whenever at least one condition is true termination may occur.

It seems that if either the interval or the exhaustive termination methods can be used for termination detection, the interval method should be used (the argument for this is given in section 4.5.6). However it not guaranteed that the interval method will always be the most efficient solution. Take the example shown in Figure 14. In this case, if the interval method is used (14a), termination will be detected long after it would be if the exhaustive method were used (14b).

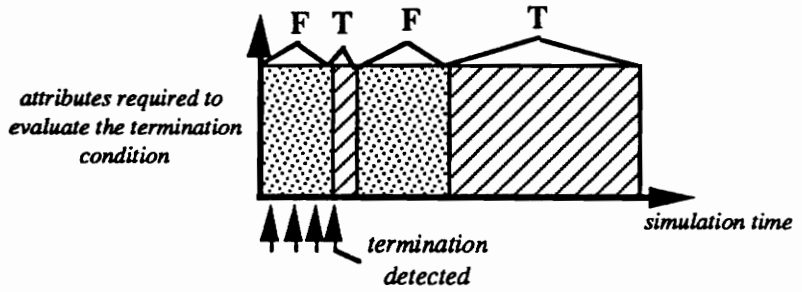
If a condition becomes true very soon after the simulation begins, the exhaustive method finds that time when termination is possible which the interval method is likely to miss. However, if there is no a priori knowledge that early termination is likely, interval termination is the better choice.

Another possible procedure to implement the disjunction of stable and nonstable termination conditions is to run both the interval and the exhaustive termination methods in parallel. Divide the available processes into two groups, and implement both algorithms in parallel. This compromise may require fewer computational steps than the worse case choice.

a. interval method



b. exhaustive method



KEY for c:

: both conditions are false

: at least one of the conditions is true

: indicates where termination condition is evaluated

Figure 14: Interval versus Exhaustive. A possible scenario where a simulation terminated by the interval termination method (a) may require more computation steps to complete than a simulation terminated by the exhaustive termination method (b) when the termination condition is the disjunct of stable and nonstable conditions.

CHAPTER 7: SUMMARY OF HOW TO IMPLEMENT DIFFERENT TERMINATION CONDITIONS

If the termination condition is stable, meaning once it becomes true it remains true, the interval termination method specified in section 4.4 may be used. If the termination condition is nonstable, the exhaustive termination algorithm specified in section 4.3 should be used.

The combination of "like" termination conditions, those that are members of the same category, prompts the use of the same method used to implement each member of the combination. For example, the conjunction of two or more stable conditions may be solved by the interval method, and the disjunction of two or more nonstable conditions may be solved by the exhaustive method. However, it is possible for the combination of two nonstable conditions to result in a stable condition, thus may be solved using the interval method.

The combination of "unlike" termination conditions presents some interesting possibilities. Solving the conjunction of stable and nonstable conditions may be done by using the "interval then exhaustive" method. The interval termination method finds the first time where the stable condition(s) is(are) true, then restores the attribute values to the last false point. The exhaustive termination method then begins searching exhaustively for a time that will satisfy the combination termination condition.

There is no clear recommendation on what method to use to detect termination for a disjunct of stable and nonstable termination conditions. The interval method will most likely find when the stable condition becomes true, however if the nonstable condition

becomes true early, the exhaustive method might solve the problem using fewer computational steps.

The summary of what method to use to implement different termination conditions is given in Table 3.

Table 3: Method for implementing each termination category

	termination category (*S* stable, *s* not)	**method for implementing**
termination conditions	*S*	interval
	s	exhaustive
termination condition conjuncts	all terms *S*	interval
	all terms *s*	exhaustive (or interval if a stable condition results from the conjunction)
	at least one term *s and* one term *S*	interval-then-exhaustive
termination condition disjuncts	all terms *S*	interval
	all terms *s*	exhaustive (or interval if a stable condition results from the disjunction)
	at least one term *s and* one term *S*	either interval or exhaustive

CHAPTER 8: CONCLUSION

Determining when to terminate an asynchronous parallel simulation using a global termination condition is inherently complex. Because each process contributing to the simulation proceeds at its own rate, at any given point in the simulation every process is likely to be at a different point in simulation time. Determining at what simulation time t the global termination condition is satisfied when each process has a different t at a given computation step and then collecting the corresponding output measures is not a trivial problem.

There were two main conclusions reached in the thesis. The first is that the time required to detect a stable termination condition (one that remains true at all future times once it becomes true) is less than the time required to detect a nonstable one. The second is that optimistic parallel simulation methods such as Time Warp [Jefferson 1985] are preferred to conservative asynchronous methods [Bryant 1977; Chandy et al. 1979] because of greater flexibility when collecting attributes.

8.1 CONTRIBUTIONS OF THIS THESIS

This thesis addresses a topic that has not been actively researched to date: how to implement a global termination condition and collect the corresponding output measures in a parallel simulation. A classification of termination conditions has been developed. Methods are described to solve the termination detection problem with termination conditions that fall into each classification category, as well as termination conditions that are a combination of conditions from different categories.

This work is of value to a simulation programmer: presenting how he could implement any termination condition for a given simulation, as well as providing heuristics to guide his choice of termination condition. If it fits a programmer's purpose equally well to use a stable termination condition or a nonstable one, he should be aware that implementing a nonstable termination condition is much more difficult and time consuming than implementing a stable termination condition.

Chapter 4 presented a formal framework to reason about simulation algorithms. Much effort was spent in making the specifications as general as possible, so that the work is applicable to all types of discrete event simulations running on any type of architecture. The ground work presented here can be reused and built upon to further simulation research.

8.2 OPEN PROBLEMS

Each of the assumptions and simplifications that were made in this paper should be resolved. The two most significant ones are the centralized assumption (that all information regarding termination be available to a single termination detector process), and the static assumption (that the number of attributes cooperating in the time warp simulation is fixed).

This thesis did not consider ways to increase efficiency when the termination condition is time consuming to solve. Abrams and Richardson [1991] suggest some possibilities, and if a simulation using a termination condition that fits into the category needs to be solved efficiently, pursuit of the topic is indicated.

It would be beneficial to have a formalization of a parallel simulation method such as Time Warp. Jefferson [1985] presented Time Warp as an example implementation, and it has never been formalized. Richardson and Abrams [1990] present a UNITY algorithm

to implement Time Warp, but no specifications or proofs. A formalization of Time Warp, especially the rollback feature, would enable formalization of the retrospective output measure collection described in section 6.

An interesting future project would be to implement parallel simulations using the termination algorithms given in this thesis, and experimentally measure the resulting performance of the different methods.

REFERENCES

Abrams, M. and D. Richardson (1991), "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation," in *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation* (Anaheim, CA, Jan.) pp 86-91.

Apt, K.R. (1986), "Correctness Proofs of Distributed Termination Algorithms," *ACM Trans Program Lang Syst 8(3),* pp 388-405.

Bellenot, S. (1990), "Global Virtual Time Algorithms," in *Distributed Simulaton 22(1)* (San Diego, CA, Jan.), pp 122-127.

Bryant, R.E (1977), "Simulation of Packet Communication Architecture Computer Systems," tech rep TR-188, M.I.T., Cambridge, MA.

Chandy, K.M. and L. Lamport (1985), "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans Comput Syst 3(1),* pp 63-75.

Chandy, K.M., V. Holmes, and J. Misra (1979), "Distributed Simulation of Networks," *Computer Networks 3,* pp 105-113.

Chandy, K.M. and J. Misra (1988), *Parallel Program Design: A Foundation,* Addison-Wesley, Reading, MA.

Chandy, K.M. and R. Sherman (1989), "Space-Time and Simulation," in *Distributed Simulation 21(1)* (Tampa, FA, Jan.), pp 53-57.

Cohen, S. and D. Lehmann (1982), "Dynamic Systems and their Distributed Termination," in *Proc ACM SIGACT-SIGOPS Symp. Principles Distr. Comput.* (Ottowa, Can.), pp 29-33.

Dijkstra, E.W. and C.S. Scholten (1980), "Termination Detection for Diffusing Computations," *Inf Proc Lett 11(1),* pp 1-4.

Dijkstra, E.W., W.H.J. Feijen, and A.J.M. van Gasteren (1983), "Derivation of a Termination Detection Algorithm for Distributed Computations," *Inf Proc Lett 16(5),* pp 217-219.

Eick, S.G., A.G. Greenburg, B.D. Lubachevsky, and A. Weiss (1991), "Synchronous Relaxation for Parallel Simulations with Applications to Circuit Switched Networks," in *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation* (Anaheim, CA, Jan.) pp 151-162.

Eriksen, O. (1988), "A Termination Detection Protocol and Its Formal Verification," *Journal of Parallel and Distr Comp 5,* pp 82-91.

Francez, N. (1980), "Distributed Termination," *ACM Trans Prog Lang & Sys 2(1),* pp 42-55.

Haldar, S. and D.K. Subramanian (1988), "Ring Based Termination Detection Algorithm for Distributed Computations," *Inf Proc Lett 29,* pp 149-153.

Jefferson, D.R. (1985), "Virtual Time," *ACM Trans Prog Lang & Sys 7,* pp 404-425.

Lakshman, T.V. and V.K. Wei (1988), "Efficient Decentralized Consensus Protocols Using Specially Structured Communication Graphs," tech mem TM-ARH-011042, Bell Communications Research, Red Bank, NJ.

Li, H.F., T. Radhakrishnan, and K. Venkatesh (1987), "Global State Detection in Non-FIFO Networks," in *The 7th International Conference on Distributed Computing Systems* (Berlin, West Germany, Sept.) pp 364-370.

Lin, Y.B. and E.D. Lazowska, "Design Issues for Optimistic Distributed Discrete Event Simulation," to appear in *IEEE Transaction on Parallel and Distributed Systems.*

Mattern, F. (1987), "Algorithms for Distributed Termination Detection," *Distributed Computing 2,* pp 161-175.

Misra, J. (1986), "Distributed-Discrete Event Simulation", *ACM Computing Surveys 18 (1),* pp 39-65.

Nance, R.E. (1981), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24* (4), pp 173-179.

Rana, S.P. (1983), "A Distributed Solution of the Distributed Termination Problem," *Inf Proc Lett 17*(1), pp 43-46.

Richardson, D. and M. Abrams (1990), "UNITY Algorithms for Detecting Stable and Non-Stable Termination Conditions in Time Warp Parallel Simulations," tech rep 90-62, VPI&SU, Blacksburg, VA.

Rozoy, B. (1988), "Termination for Distributed Systems with Asynchronous Message Passing: Model and Cost," *Computers and Artif Intell 7(1),* pp 1-23.

Spezialetti, M. and P. Kearns (1986), "Efficient Distributed Snapshots," in *Proc of the Sixth Conference on Distributed Computing Systems,* pp 382-388.

Topor, R.W. (1984), "Termination Detection for Distributed Computations," *Inf Proc Lett 18(1),* pp 33-36.

APPENDIX: AN OVERVIEW OF UNITY

If a problem is formally and accurately specified, insights into problem solutions can be more easily made. A specification method may include helpful guidelines for specification refinement, as well as a system to prove that each refinement implies the preceding specification.

In order to formalize the termination problem and develop some solutions, this thesis employs the UNITY notation of Chandy and Misra [1987]. UNITY provides a computation model, a specification notation, and a proof system to verify the correctness of specification refinements. UNITY provides methods to compose larger programs from smaller ones, which is useful for our purpose because we can separately describe the termination algorithm and the underlying, nonterminating simulation algorithm, and then compose the two parts into a terminating simulation algorithm. The algorithms presented in this thesis are architecture independent; UNITY provides heuristics to guide refinement of these algorithms into efficient programs for various classes of target architectures.

Chandy and Misra [1987] describe a UNITY program as follows:

> A program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following "fairness" rule: Every statement is selected infinitely often. (p. 9)

UNITY Program Structure

UNITY programs are composed of four sections. The **declare** section names the variables used in the program and their types. The **always** section is used to define certain variables as functions of others (this section is optional to use in a UNITY program, but often convenient). The **initially** section is used to define initial values of the variables. The **assign** section contains a set of assignment statements. The following simple program contains all four sections.

Program *simple_example*

declare x,y : integer
always $y = 2 * x$
initially $x = 1$
assign $x := x + y$ if $x < 10$

end

The program *sort2* given next shows a more involved **assign** section (taken from Chandy and Misra [1988], page 32).

Program *sort2*

declare i, N : integer
A : array $[0..N]$ of integer
assign $\langle \| i : 0 \le i < N \wedge \text{even}(i) :: A[i], A[i+1] := A[i+1], A[i]$ if $A[i] > A[i+1] \rangle$
$\square \langle \| i : 0 \le i < N \wedge \text{odd}(i) :: A[i], A[i+1] := A[i+1], A[i]$ if $A[i] > A[i+1] \rangle$

end

UNITY Notation

The following paragraphs explain UNITY terms that are used in this thesis.

Computation State: The computation of a UNITY program may be described by asserting what predicates hold for particular computation states. The computation state is simply represented by the value of all variables.

After execution of an assignment statement, a new computation state is reached. Statement execution is atomic with respect to computation states: no new state exists until after the *entire* statement is executed.

Fixed point (FP): The notion of program termination is not an explicit part of UNITY's computational model. A program is said to reach *fixed point* (FP) when all the values on the left and right sides of each assignment statement in the program are identical, so that it makes no difference whether execution continues or not. Therefore, it is of fundamental importance to prove that the termination algorithms proposed in this thesis reach FP.

Superposition: Superposition allows structuring a program as a set of "layers"; a layer can draw on the services provided by its lower layers (though it is not a symmetric relationship: the lower layers are not allowed access to the upper layers).

The guidelines for superposition given by Chandy and Misra are extended for the purposes of this thesis, for they provide no way to add a conditional onto a lower layer. We extend superposition to allow this, for it is the most straightforward way to describe what must be done to the underlying nonterminating simulation program to allow implementation of a termination condition. If the conditional "if $\neg$*terminate*" is superposed onto each assignment statement in the underlying program, *terminate* implies that the

program has reached FP, for no statement could be executed. The variable *terminate* is stable in the underlying program, so all basic properties of the program (excepting nontermination) are preserved.

Union: The combination of two *component* programs to form a *composite* program is called a union operation in UNITY. Union is useful in expressing a complex problem. When two programs are combined together, the corresponding sections of the two programs are merged (concatenate the two define sections, concatenate the two initially sections, etc.). The union mechanism allows UNITY programs to employ the software engineering principle that a large program should be composed from a number of smaller component programs, where each component is developed, understood, and proven correct in isolation.

The symbol ▯: This symbol is used to separate assignment statements. For example, "r := x ▯ s := y" implies that computation proceeds by executing any one of the two assignments, selected nondeterministically. The selection obeys the fairness rule that every assignment is executed infinitely often.

Structure of the Assignment Statement: An assignment statement consists of one or more assignment components separated by "||". There are two kinds of assignment components: the enumerated assignment and the quantified assignment.

> **Enumerated Assignment:** An enumerated assignment assigns the values of expressions on the right of ":=" to corresponding variables listed on the left of ":=". For example, "x, y, z := z, x, y" is equivalent to "x := z || y := x || z := y". This denotes the traditional multiple assignment: all expressions in the right side of the enumerated assignment are evaluated before assignments are made to the right.

Quantified Assignment: A quantification has a scope delineated by "<" and ">", and is of the form "*variable-list* : *boolean-expr* ::". The variables in the variable list are called *bound* variables. An *instance* of a quantification is a set of values of bound variables that satisfies the boolean expression. A quantified assignment denotes zero or more assignment components that are obtained by replacing bound variables by their instances. For example, to initialize a given array $A[0..N]$ to zero: $< \| \; i : 0 \leq i \leq N \; :: \; A[i] := 0 \; >$. Two more quantified assignment statements are given in the example program *sort2* given previously.

Quantified Expression: UNITY allows the use of quantification (explained in the preceding paragraph) in writing expressions. Such an expression is <*op quantification expr*>. "The syntactic units *expr* and *op* denote expression and operator, respectively. The value of the expression defined this way is the result of applying the operator, *op*, to the set of expressions obtained by substituting the instances of the bound variables in the inner expression." For example, $<\min \; i : 0 \leq i \leq N :: A[i]>$ denotes the smallest element in $\{A[0],...,A[N]\}$.

Leads to ($\mapsto$): Many progress properties of programs are stated using *leads to*. This symbol is rather self-explanatory: if the expression on the left side of the symbol holds, then the expression on the right will eventually hold. For example, given predicates p and q, $p \mapsto q$ denotes that once p becomes *true*, q is or will be *true*.

Ensures: The expression p *ensures* q shows that if p is *true* at some point in the computation, p remains *true* as long as q is *false*, and eventually one statement in the program will cause q to become *true*.

Stable: A stable predicate remains *true* once it becomes *true*. FP is a stable property.

Invariant: A predicate q is invariant if the initial condition of the program implies q, and q is stable.

Detects: For a given program, p *detects* q means $(p \Rightarrow q) \wedge (q \mapsto p)$. That is, $p \Rightarrow q$ is an invariant of the program, and if q holds then p holds eventually.

***p* in *program_name*:** This notation is useful when more than one program is being described. For example, the expressions "**stable** p in *prog1*" and "$p \mapsto q$ in *prog2*" are convenient to infer that the property of a stable p holds for the first program and the leads to property holds for the second.

Sequence Notation: A sequence of items is denoted by enclosure within << and >>. The expression $x \sqsubseteq y$ denotes that x is a prefix of y. If S denotes a sequence, and $S = <<x_0,x_1,...x_N>>$, then $head(S) = x_0$, $tail(S) = <<x_1,..x_N>>$, and $last(S) = x_N$. If $S = <<x_0,x_1,x_2,...>>$, then $head(S) = x_0$, $tail(S) = <<x_1,x_2,..>>$, and $last(S)$ is undefined.

VITA

Debra Sue Richardson was born in Harrisburg, Pennsylvania on October 14, 1964. Within a few years her family moved to a farm in Albemarle County in central Virginia; Debbie graduated from Western Albemarle High School. Possessing a teenager's motivation to leave the nest, she spurned the nearby University of Virginia and enrolled at Virginia Tech at the age of 16. After graduating with a B.S. in Animal Science and a teaching certificate, Debbie spent the following two years teaching high school Chemistry, Biology, and Physical Science at Fauquier High in northern Virginia. Two years of being abused by teenagers and not being able to pay the bills prompted a desire for more (and different) education. Night and summer school at Northern Virginia Community College and the Northern Virginia Graduate Center of Virginia Tech enabled Debbie to return to Blacksburg and become a graduate student in Computer Science. After completion of her Master's, Debbie will move to Lexington, Kentucky to begin a job with IBM that should enable her to pay the bills.