Terminating Parallel Simulations

Marc Abrams

TR 92-01

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

February 4, 1992

Terminating Parallel Simulations

Marc Abrams
Computer Science Department
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
U.S.A.
abrams@cs.vt.edu

TR-01 January 1992

Abstract

The simulation termination problem consists of three parts. First, find a value of simulation time, denoted t, such that a termination condition evaluated using simulation model attributes at time t has value *true*. Second, report the value of each simulation output measure at time t. Finally, stop execution of the simulation. The problem is more difficult than the classical termination detection and stability detection problems for two reasons. First, a simulation model termination condition may not be a stable property of the computation. Second, evaluating the termination condition asynchronously with respect to the simulation implies that when termination is detected the simulator has already modified old attribute values needed to compute output measures. The paper presents two different general solutions to the simulation termination problem for various simulation protocols and parallel architectures. Implementation of both solutions with conservative-synchronous, optimistic, and conservative-asynchronous simulation protocols as well as synchronous shared-memory, distributed, and asynchronous shared memory architectures is discussed.

1 Introduction

Simulation models use many different rules to decide *when* to terminate. However, most parallel discrete-event simulators can handle only a single termination condition: each logical process constituting the simulation terminates when its local simulation time equals or exceeds a value T. To apply parallel simulation protocols to any simulation model, algorithms must be developed to use any arbitrary termination rule, which is the objective of this paper.

The simulation termination problem is to devise an algorithm that can be added to a *non-terminating* simulation program so that the modified program will find a simulation time at which a *termination condition* (described later) holds, then calculate the value of each simulation output measure using attribute values at this simulation time, and finally shut down the simulation program (i.e., terminate the processes comprising the simulation).

This problem formulation is based on the belief that in a commercial parallel simulation system, one would like the user to specify a simulation model without worrying about termination, and then separately specify various termination conditions over the multi-year lifetime of the simulation program. The simulation system should automatically compose the termination condition with the non-terminating simulation, and the problem statement is to develop a method of composition. Composing a termination condition with a non-terminating simulation appears preferable to a termination mechanism that is "built in" to a simulation program, which complicates maintenance of the program.

A *termination condition* is a boolean valued function whose domain is a subset of all simulation model attributes. Time is only one attribute upon which a termination condition may be based, as the following examples illustrate:

T1: In a multiprogrammed computer system simulation, has deadlock occurred?

T2: In a colliding pucks simulation, have *at least* M collisions occurred in the system?

T3: In a colliding pucks simulation, have *exactly* M collisions occurred in the system?

T4: In a colliding pucks simulation, have $M \pm 0.1M$ collisions occurred in the system?

T5: In a mobile telephone simulation, is this the first time that three telephones' radii of communication overlap?

T6: In a mobile telephone simulation, is this the tenth time that three telephones' radii of communication overlap?

T7: In a mobile telephone simulation, is this the last time that three telephones' radii of communication overlap in the simulation time interval [0,150]?

T8: Let δ and ϵ be constants. In an open queueing network simulation, does the average number of jobs in the system at times t and $t - \delta$ differ by less than ϵ or have at least M jobs entered the system?

T9: In an electrical circuit simulation does the voltage at a set of nodes meet some stability condition (Evaluating the stability condition requires a costly numerical integration of a function.)?

T10: Has the instantaneous number of jobs waiting for service in a queueing network simulation reached its maximum over the simulation time interval [0,1000]?

1.1 Categorization of Termination Conditions

Simulation termination conditions can be categorized based on three attributes:

Stability: A termination condition is *stable* if, once it holds, it continues to hold.[3] Conditions T1 and T2 are the only stable properties listed above. If a termination detection algorithm evaluates a stable termination condition using the values of simulation model attributes at simulation time t, and the condition is *false*, then the condition must be false at all times smaller than t. In contrast, no such inference can be made in general for non-stable conditions; therefore stable conditions are inherently easier to detect than non-stable conditions.

Time quantification: Viewing a termination condition as a predicate calculus formula, expression of a termination condition may or may not require quantification over time. Conditions T5–T6 and T10 require quantification over time. For example, consider condition T5. Letting $\hat{C}(t)$ denote the Boolean predicate "three telephones' radii of communication overlap," the termination condition is $\hat{C}(t) \wedge \langle\ \nexists t' : t' < t :: \hat{C}(t')\ \rangle$.

Time quantified conditions require the condition to be evaluated for a set of times, rather than a single time, which makes them inherently harder to detect than unquantified conditions. In fact, the termination condition might have to be evaluated at *every* simulation time at which a model attribute changes value, such as in condition T10.

This paper considers stable and non-stable non-quantified termination conditions; a general solution to quantified conditions is left as an open problem. (The Exhaustive Termination algorithm of Abrams and Richardson [1] and the Algorithm TW2 of Lin [6] both detect one *specific* quantified condition, namely the first time that a Boolean condition holds, such as condition T5. However the general problem cannot be solved by any existing algorithm.)

Termination conditions T2 through T10 illustrate various difficulties in detecting simulation termination. All ten conditions require data that is global to the processes that comprise a parallel simulation. Recording the global data that is required by a termination condition can limit the performance of parallel simulation.

Conditions T3, T5, and T8 are difficult to detect because they only hold for narrow intervals of simulation time. Condition T4 is somewhat easier to detect than T3 because it holds for a wider interval of simulation time. Condition T9 suggests that sometimes the termination condition may be as costly as or more costly than the simulation itself to evaluate; in this case parallel evaluation of the termination condition may be more critical than parallel execution of the simulation model. Some conditions represent the conjunct and/or disjunct of more primitive conditions, such as conditions T8.

At worst, detecting a termination condition requires evaluating the condition at every simulation time at which a logical process changes the value of any simulation attribute in the domain of the termination condition. Detecting a quantified termination condition additionally requires comparison of the value of measures at multiple time simulation times. Therefore a parallel simulation could require more wall clock time to execute than a sequential simulation. The problem may not have a simple solution; for example Lin has shown that adding extra processors to an optimistic protocol to evaluate a termination condition may introduce more overhead than benefit.[6] This may pose a fundamental limitation on the speedup of parallel simulation.

1.2 Framework for Solutions

This paper is intended to provide a theoretical basis for the simulation termination problem and to provide a framework for future development of solutions. Therefore we formally define and develop two general solutions to the simulation termination problem. The framework fits all proposed termination algorithms (Lin and Lazowska, Abrams and Richardson, Richardson, Sanjeevan and Abrams, and Lin.[7, 1, 8, 6]), as discussed in Section 4.2.

Our solutions are general in the sense that they work with any underlying discrete-event simulation, computer architecture, world view, and time flow mechanism. The solutions are efficient for any parallel simulation protocol that tends to fill in space-time in ascending order.

We adopt the view of separating correctness concerns from efficiency concerns.[3, p. 9] Therefore our solutions state *what* tests and assignments must be executed to solve the termination problem. However they leave unspecified the issue of *how* the solutions are mapped to a parallel computer architecture. In particular, they leave unspecified the issue of whether the termination detection algorithm is mapped to separate processes and processors from the underlying simulation, because this is a question of efficiency that must be reexamined for each combination of computer architecture and parallel simulation protocol that are used. The "how" question is addressed in Section 5.

The problem statement and termination algorithms in the following section are presented in two forms, first an informal and then a formal description. Furthermore, the algorithms are presented as a sequence of three refinements. The informal presentations are intended to give sufficient detail to permit a reader to skip the formal sections but still understand the algorithms. The formal presentations eliminate ambiguities in the informal presentations and allow rigorous proof of both progress and safety properties.

1.3 Relation to the Classical Termination Problem

The parallel simulation termination detection problem is related to two problems that have been studied extensively in the literature: termination detection [4, Ch. 9] and stability detection [4, Ch. 11]. The termination detection problem is to identify when a set of processes has ceased its computation, which requires detecting that each process is idle and that no process will ever again become active. The stability detection problem is to identify when a stable property holds for an ongoing computation. Deadlock in a system that does not break deadlocks is an example of a stable property.

The parallel simulation termination problem is more difficult than the classical termination detection problem. The classical termination problem is one of determining when all processes have become permanently idle. In contrast the underlying simulation program is assumed to continue execution until stopped by the superposed termination detection algorithm, just as in the stability detection problem "The superposed program is required to detect a property of an *ongoing* underlying computation."[3, p. 270] Second, the classical termination problem is one of detecting a termination condition that is a conjunct of predicates, where each predicate can be evaluated by one process using only variables private to that process. In contrast, a simulation termination condition is an arbitrary function of any or all simulation attributes, including those global to all processes comprising the simulation.

The parallel simulation termination problem differs from the stability detection problem in three ways. First, simulation termination requires identifying a point in the computation at which a possibly non-stable condition holds, whereas the classical detection problem assumes a stable condition. Detecting non-stable conditions is difficult because it can require exhaustive examination of all global system states; therefore a good solution to the simulation termination problem should identify ways to deduce that certain global states may be excluded from consideration, as is done in Section 4. Second, in the simulation termination problem, when a time satisfying the termination condition is identified, a set of output measures must be evaluated. The values of simulation model attributes required to evaluate output measures, generally distinct from those required to evaluate termination, must be available. Further, the attribute values at the termination time, which lies in the past of the processes that are carrying out the parallel simulation, must be used. No analog to recovering old attribute values to evaluate output measures exists in the stability detection problem. Third, no analog to time-quantified termination conditions has been treated in the stability detection literature.

1.4 Summary of UNITY

The formal presentations in this paper use the notation, computation model, and proof system of UNITY.[3] We use UNITY for several reasons. First, UNITY offers a way to state the algorithms in a computer architecture and programming language independent form, which facilitates our goal of developing a framework for all possible solutions to the termination problem for any simulation model and parallel execution method. Second, UNITY allows an algebraic specification of the algorithm, which reduces our tendency to introduce arbitrary implementation decisions into our solution. Finally, UNITY provides the ability to state and prove both progress and safety properties in relatively compact form. In particular, the *detects* logical relation simplifies formal reasoning. A summary of UNITY follows; a lengthier tutorial appears in Abrams, Page, and Nance.[2]

Computational Model: "A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple assignment statements."[4, p. 9] The *state* of a program after some step of the computation is the value of all program variables. In the UNITY computation model, "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:" at any point during program execution, every statement in the program must be executed at some point in the future.[4, p. 9]

A UNITY program never terminates. However, a program may reach fixed point (FP), which is a computation state in which execution of any assignment statement does not change the state. At FP, the left and right hand side of each assignment statement are equal, and an implementation can thereafter terminate the program.

UNITY Specifications: In addition to expressing programs, UNITY can be used to state and reason about program specifications through a set of logical relationships defined below. (UNITY denotes $\forall i, i \in W, C(i)$ by $\langle\ \forall i : i \in W :: C(i)\ \rangle$.)

Let p and q denote arbitrary predicates, or Boolean valued functions of the values of program variables. Let s denote an assignment statement in a program. The assertion $p \Rightarrow q$ is read "if

p holds then q holds." The assertion $\{p\}s\{q\}$ denotes that execution of statement s in any state that satisfies predicate p results in a state that satisfies predicate q, if execution of s terminates.

We will use three logical relations of UNITY: *unless*, *leads-to*, and *detects* . The definitions below are those of Chandy and Misra.[4, Ch. 3]

Unless: For a given program F, "p *unless* q" means that if p is true at some point in the computation and q is not, in the next step (i.e., after execution of a statement) either p remains *true* or q becomes *true*. Therefore either q never holds and p continues to hold forever, or q holds eventually (it may hold initially when p holds) and p continues to hold at least until q holds. Formally, p *unless* $q \equiv \langle \forall s : s \text{ in } F :: \{p \wedge \neg q\}\, s\, \{p \vee q\}\rangle$.

Two special cases of *unless* are **stable** and **invariant**. A stable predicate – just like a stable termination condition – continues to hold once it holds: **stable** $p \equiv p$ *unless false*. An invariant property always holds in any program execution: **invariant** $p \equiv$ (initial condition $\Rightarrow q$) $\wedge$ **stable** q.

Leads-to: Leads-to is denoted by the symbol $\mapsto$. The assertion "$p \mapsto q$" means that if p becomes true at some point in the computation, q is or eventually will be *true*. The formal definition of leads-to is somewhat lengthy, and is not given here.

Detects: The assertion "p *detects* q" means that if q holds at some point in the computation, then eventually p will also hold; further more when p holds q also holds. Formally p *detects* $q \equiv (p \Rightarrow q) \wedge (q \mapsto p)$. The simulation termination problem is expressed in terms of *detects* in Section 3.

Program notation: Figure 1 shows the syntax of UNITY program. A <*var–decl–list*> is a list of variable declarations in Pascal. The <*initial–list*> specifies the initial values of program

Program <*name*>
  declare <*var-decl-list*>
  initially <*initial-list*>
  always <*always-list*>
  assign <*asg-list*>
end { <*name*> }

Figure 1: UNITY program syntax.

variables. The <*always-list*> contains equations that may be thought of as functions. The <*asg-list*> contains a list of assignment statements separated by the symbol "□". An assignment statement may be multiple; the symbol "||" separates the component assignments. For example, $x := y \parallel y := x$ swaps x and y. An assignment statement may be conditional; if the condition is not met then the assignment is the identify function. For example, $x := -x$ if $x < 0$ assigns the absolute value of x to x. A multiple assignment may be quantified. For example, given array A[0..N], the statement ⟨|| i : 0≤i<N :: A[i]:=A[i+1] ⟩ shifts A[i+1] to A[i] for the specified range of i in parallel.

Theorems: The theorems listed below are used in the proofs. Each proof is written in the form of $\frac{\text{hypothesis}}{\text{conclusion}}$.

$$\frac{p \Rightarrow q}{p \mapsto q} \tag{1}$$

$$\frac{p \mapsto q, r \textit{ unless } b}{p \wedge r \mapsto (q \wedge r) \vee b} \tag{2}$$

$$\frac{p \mapsto q, r \text{ is stable}}{p \wedge r \mapsto q \wedge r} \tag{3}$$

$$\frac{\langle\ \forall m\ :\ m \in W\ ::\ p(m)\ \mapsto\ q(m)\ \rangle}{\langle\ \exists m\ :\ m \in W\ ::\ p(m)\ \rangle\ \mapsto\ \langle\ \exists m\ :\ m \in W\ ::\ q(m)\ \rangle} \tag{4}$$

$$\frac{p \mapsto q, p' \mapsto q'}{p \vee p' \mapsto q \vee q'} \tag{5}$$

For any set W:

$$\frac{\langle\, \forall m : m \in W :: p \wedge (M = m) \mapsto (p \wedge M \prec m) \vee q \,\rangle}{p \mapsto q} \tag{6}$$

$$\frac{p \textit{ detects } q, q \textit{ detects } r}{p \textit{ detects } r} \tag{7}$$

Theorems (1), (2), (3), (4), (5), (6), and (7) are the implication theorem, the progress safety progress theorem, a corollary to the progress safety progress theorem, the general disjunction theorem, the finite disjunction theorem, an induction theorem for leads-to, and the transitive property for *detects* , respectively.[3, pp. 64, 65, 71, 65-66, 71, 72, 210]

Theorem (6) is used to prove $p \mapsto q$ by induction over the value of a *metric*. The metric is denoted by M, which is a function of a program state. The range of M, set W, must be partially ordered under $\prec$ and have a lower bound. The inductive hypothesis of Theorem (6) is that if one can prove that in any program state either $p \wedge \neg q$ holds and the value of the metric eventually decreases or q holds, then because the metric has a lower bound, q must eventually hold.

We will often prove a *detects* relation by decomposing it into a chain of more primitive *detects* , then proving the primitive *detects* using the definition of *detects* and other theorems, and finally applying Theorem (7).

In addition to these theorems, subsequent proofs require an instance of the *Substitution Axiom*: if "**invariant** I" holds, then any predicate p may be replaced by $p \wedge I$.

Finally, proofs are written in a structured form consisting of a sequence of deductions; each deduction is followed by a comma and a justification for that deduction.

2 Specification of Simulation

Before we can present a formal statement of and solution to the simulation termination problem, we need a formal statement of what it means to carry out a discrete event simulation. Therefore we start by formally specifying simulation itself to the extent needed for subsequent sections. The definition will be presented informally, and then formalized as a set of UNITY assertions.

Because we seek a general solution to the simulation termination problem, our definition of simulation must subsume both sequential and parallel execution. It must also subsume all conceptual frameworks used for simulation, such as event-, activity-, and process-oriented world views. It must also subsume all time flow mechanisms used in discrete-event simulation, including next event, process view, transaction view, and three-phase approach.

We propose the following definition of "pure" simulation. A simulation represents the time evolution of a set of attributes. A discrete-event simulation, viewed operationally, starts with each attribute assigned an initial value. At a sequence of time instances, each attribute will be assigned a value. The value assigned to an attribute may be the same as or different from its old value.

Let $a_1, a_2, \ldots, a_i, \ldots, a_n$ represent the attributes modeled by a simulation program. A simulation program starts with an initial value for each of the n attributes and calculates a sequence of values for all attributes. Integer i, for $1 \leq i \leq n$, indexes an attribute. All UNITY formulas in which i appears are implicitly quantified over all $i \in [1, n]$, unless otherwise noted.

The integer k, for $k = 1, 2, 3, \ldots$, indexes *simulation time steps* in the simulation. The first simulation time step, $k = 1$, denotes the initial assignment of values to attributes. The k-th simulation time step denotes the k-th assignment of values to model attributes.

Suppose that simulation time step $k = 1$ corresponds to simulation time 0.0, time step $k = 2$ to simulation time 340.35, time step $k = 3$ to simulation time 990.8, and so on. The values $0.0, 340.35$, and 990.8 are irrelevant in reasoning about simulation termination. Therefore we only refer to time steps 1,2, and 3, and do not define any notation to refer to the simulation times $0.0, 340.35$, and 990.8. The "current simulation time" is simply one of the n attributes in a simulation model.

Let N denote an upper bound on the number of values that are assigned to the attribute set. Equivalently, N is the maximum number of time steps that are simulated before the underlying simulation stops execution possibly without the termination condition being satisfied. In theory, there is always an integer N that serves as an upper bound on the number of time steps that a simulation can compute. In practice, the limit N alway exists, because no computer program executes forever. Note that the limit N has nothing to do with the termination problem; a person may write a simulation program in which the termination condition is never met, in which case the simulation will not compute more than N time steps. Otherwise the simulation generally detects termination in less than N time steps.

We specify simulation in terms of n arrays, $a_i[1..N]$, for $i = 1, 2, \ldots, n$, and variables K and *TCD.*

Array $a_i[1..n]$*:* Array elements $\langle\ \forall i :: a_i[1]\ \rangle$ represent the initial values of the n model attributes. Array elements $\langle\ \forall i : 1 \leq k \leq N :: a_i[k]\ \rangle$ represent the k-th values assigned to model attributes.

K: Variable K is initially 1. At any point during simulation, the values assigned to the first K time steps of all attributes will never change. Attributes at time steps $1, 2, \ldots, K$ are termed *committed*; the set of uncommitted time steps in a simulation is $\{k : K < k \leq N\}$. Therefore K represents a horizon of committed time steps that advances during simulation. Attribute values at committed time steps have been assigned their final values, while attributes at uncommitted time steps may or may not have been assigned values, or their values may be reassigned as the simulation progresses. We will not specify any constraints on how attributes at uncommitted time steps are assigned values, so that the solution presented admits any parallel execution scheme. The distinction of committed and uncommitted time steps makes the solution presented most efficient for simulation methods that tend to fill in space-time in ascending order. (For example, optimistic methods reassign attribute values as each logical simulation process advances, and rolls back, and again advances.)

TCD: Boolean simulation program variable *TCD* (i.e., "termination condition detected") is initially false. The simulation never assigns a value to this variable. An external algorithm (namely the termination detection algorithms proposed in Section 4) will set *TCD* to *true* when it detects termination based on the final attribute values calculated by the underlying simulation.

Let U denote the underlying simulation program. Specification *Sim* contains the formal specification of U. The term " in U " at the end of each assertion means that the assertion applies in program U, but not in any other program, such as the termination detection program developed in successive sections.

Specification *Sim*: Initially, $TCD = false$ and $K = 1$.

(Sim1) $\langle \forall j :: K = j \textit{ unless } K > j \rangle$ in U

(Sim2) $\langle \forall k :: \neg TCD \wedge K < N \wedge K = k \mapsto K > k \rangle$ in U

(Sim3) **invariant** $1 \leq K \leq N$ in U

(Sim6) **stable** $\langle \forall k, x : 1 \leq k \leq K :: a_i[k] = x$ in $U \rangle$

(Sim7) **invariant** $TCD \Rightarrow FP$ in U

Specification *Sim* states the following. Initially the simulation has not detected a termination condition and only the initial simulation time step is committed. The committed time horizon increases monotonically (Sim1); this precludes the possibility of a committed time step becoming uncommitted. If in the present computation state the simulation the termination condition has not been detected and there still exist uncommitted time steps, then eventually at least one more time step commits (Sim2). The committed time horizon never exceeds N, the limit on simulation duration (Sim3). At any point during the simulation we are guaranteed that attributes $\langle \forall i, k : 1 \leq k < K :: a_i[K] \rangle$ have been assigned their final values (that is their values never change) (Sim6). The underlying simulation has reached fixed point when variable TCD is true; therefore the underlying simulation can be terminated (Sim7).

Note that the specification does not state *how* a new value is calculated for an attribute. This is unnecessary to specify the simulation termination problem and its solution; in fact we want our solution to be correct no matter how attributes are updated. The specification only states that the simulation make progress because K is guaranteed to increase as long as the termination condition has not been detected and the simulation has not reached time step limit N (Sim2). The specification also does not address the issue of *when* variable TCD is assigned the value *true*; this is part of the simulation termination problem.

3 Problem Statement

3.1 Informal Description

Let C denote a termination condition, and let $C(k)$ denote the condition value when evaluated using the values of simulation model attributes at the k-th time step, for $k \geq 0$. We write $C(k)$ for convenience, even though C is also a function of a subset of the attribute set $\langle \forall i :: a_i[k] \rangle$. We also define $C(k) \equiv \mathit{false}$, for $k \leq 0$.

The simulation termination problem is the following. Construct a termination detection program, denoted *TD*, that will:

P1: find a simulation time step at which C holds, and

P2: calculate the value of each simulation output measure using simulation attribute values at the time step found in P1.

Furthermore, the composition of TD with the underlying simulation, Sim, should reach fixed point when P1 and P2 are complete.

For simplicity, assume that a simulation model calculates a single output measure; generalizing the solution to multiple output measures is straightforward. Function $om(k)$ denotes the output measure; its value is the output measure evaluated using attribute values at the k-th time step. We write $om(k)$ for convenience, even though om is also a function of a subset of the attribute set $\langle \forall i :: a_i[k] \rangle$. In general functions $om$and C require different subsets of the attributes.

Program Composition: Programs TD and U will be composed using the UNITY *union*, denoted $U \Box TD$[4, Chapter 7.2]. Program $U \Box TD$ consists of appending programs TD and U together.

The properties in specification Sim are stated to hold only in program U. The variables used in Sim ($a_i[1..n]$ and K) will not be modified in program TD. Therefore (Sim1), (Sim3) and (Sim6), which only use these variables, also apply in program $U \Box TD$, by the locality corollary of the Unity union theorem[4, p. 157].

3.2 Formal Description

Parallel simulation termination problem: Given an underlying simulation program, U, meeting specification *Sim* and a termination condition C, devise a termination detection program, *TD*, introducing variable *om* local to *TD* such that:

PS1: *TCD detects* $\langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ C(k)\ \rangle$ in *TD*,

PS2: **invariant** $TCD \Rightarrow \langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ om = om(k)\ \wedge\ C(k)\ \rangle$ in *TD*, and

PS3: **invariant** $TCD \Rightarrow FP$ in $U \square TD$

The *detects* relation (PS1) implies the following. If $C(k)$ holds for some committed time step k, then eventually the simulation will reach fixed point, when it can be terminated. In addition, when fixed point holds, there will still exist a committed time step k for which $C(k)$ holds. Formula PS2 states that when fixed point holds, variable *om* contains the desired output measure, evaluated using attribute values from a time step at which the termination condition holds. Finally, assertion PS3 states that the composite program has reached fixed point and can be terminated when *TCD* is assigned *true*.

The following lemma about C is a property of program $U \square TD$, no matter what termination detection program is used: the value of the termination condition at every committed time step is constant. The property follows from the fact that all attributes are assigned their final attribute values at all committed times in the underlying simulation.

Lemma 1

$$\textbf{stable}\ \ \langle\ \forall k, x\ ::\ 0 < k \leq K\ \wedge\ x \in \{\text{true}, \text{false}\}\ ::\ C(k) = x\ \rangle\ \textit{in}\ U \square TD$$

Proof: Follows from (Sim1), (Sim6), and the definition of C. □

4 General Solutions

This section presents three solutions to the termination problem: *Soln1*, *Soln2*, and *Soln3*. In general we expect that the wall clock time required to execute the underlying simulation, *U*, is elongated by some period of time due to the termination detection program, *TD*. The solutions provide two different strategies of reducing time required to evaluate the termination condition. The first, in *Soln1* and its refinement, *Soln2*, attempts to minimize the time added by *TD* by evaluating the termination condition at different time steps in parallel. The second, in *Soln3*, attempts to minimize the added time by evaluating the termination condition fewer times, by guessing time steps at which the termination condition is *true*, possibly at the expense of increasing the number of time steps simulated by the underlying simulation. This is equivalent to defining a function that specifies in what order time steps should be tested. We view the two techniques – parallel evaluation and ordering – as mutually exclusive, because parallelism is enhanced by the absence of ordering.

All assertions that follow hold in program *TD*, unless otherwise noted.

4.1 Reducing Termination Time through Parallel Evaluation

4.1.1 Overview

Solution strategies *Soln1* and *Soln2* only evaluate the termination condition at committed time steps. Committed time steps will be partitioned into three subsets:

unknown-if-false (UF): the set of all committed time steps at which the termination condition is not known to fail,

known-false (KF): the set of all committed time steps at which the termination condition is known to be *false* through evaluation of the termination condition, and *irrelevant (I)* a set of all committed time steps which are unnecessary to evaluate, as explained below.

The preliminary solution (Section 4.1.2) proposes a strategy that only categorizes time steps as unknown-if-false and known-false. The subsequent refinement adds rules to identify irrelevant time steps by defining a notion of congruent time steps to identify a minimal set of time steps at which the termination condition must be evaluated.

4.1.2 A Preliminary Solution

Informal description: We propose a strategy which employs local variables UF, KF, and s in the termination detection program, TD. The proposed solution always maintains the invariant that sets UF and KF mutually exclusively and exhaustively partition the set of all committed time steps and zero. Variable s contains a time step.

Variable s is initially zero, a time step at which the termination condition fails by definition. Whenever the termination condition fails at time step s and there exists some unknown-if-false time step not equal to s, the time step is added to the set of known-false time steps, in KF, and s is assigned some unknown-if-false time step, from UF. We leave unspecified the *order* in which we examine values in the interval to permit mapping the solution to a parallel architecture, as described in Section 5.1.1.

Formal specification of solution strategy: Initially, $KF = \emptyset$, $UF = \{0\}$, and $s = 0$.

Specification *Soln1*: [(td1)- (td7)]

(td1) *TCD detects* $C(s)$

(td2) **invariant** $TCD \Rightarrow om = om(s)$

(td3) **invariant** $s \in UF$

(td4) **invariant** $UF \cup KF = \{0, 1, 2, \ldots, K\} \wedge UF \cap KF = \emptyset$

(td5) $\langle \forall k :: s = k \wedge \neg C(s) \wedge \|UF\| > 1 \mapsto k \in KF \rangle$

(td6) **stable** $\langle \forall k : 0 \leq k \leq N :: k \in KF \rangle$

(td7) **invariant** $\langle \forall k : k \in KF :: \neg C(k) \rangle$

(td8) **invariant** $TCD \Rightarrow FP$

Specification *Soln1* states the following. If during simulation variable s is assigned a time step at which the termination condition holds, then $C(s)$ will continue to hold until variable *TCD* is set to *true* to detect termination (td1). When variable *TCD* is set to true, variable om contains the value of the output measure using attribute values at time step s (td2). Variable s always contains an unknown-if-false time step (td3). The set consisting of all committed time steps and the integer zero is partitioned into sets *UF*, of unknown-if-false time steps, and *KF*, of known-false time steps (td4). (Assertions (td3) and (td4) imply that the termination detection algorithm never evaluates the termination condition at any time step for which final values of attributes are not known.) If the termination condition is *false* at time step s and there is an unknown-if-false time step other than s, then eventually the value of s is added to set *KF* (td5). (Assertions (td3) and (td5) imply that when the value of s is added to set *KF*, then s is assigned another unknown-if-false time step.) Time steps are never removed from the set of known-false time steps (td6). (Assertions (td4) and (td7) imply the following. If in some computation state $K = k$, for some integer k, and subsequently the underlying simulation advances K as simulation time steps commit, then the values $k+1, k+2, \ldots, K$ must be added to set *UF*. The termination condition does not hold at any *KF* time step (td7). The termination detection program has reached fixed point, and hence can be terminated, when *TCD* is *true* (td8).

Correctness proof of solution strategy: We prove that the three assertions (PS1 through PS3) comprising the problem specification are met by any solution strategy that satisfies conditions (td1) through (td8) of *Soln1*.

The proofs use of induction (Theorem (6)) and the transitivity of *detects* (Theorem (7)). Induction is used to reason about repeated assignment to variable s. The metric (i.e., function of current program state) required by Theorem (6), M, is an upper bound on the number of times that the termination condition will be evaluated for the remainder of the simulation. The upper bound has the property that it decreases in value each time s is assigned a new value. The metric is the sum of the number of unknown-if-false time steps ($||UF||$) and the number of uncommitted time steps that the simulation has or will compute ($N - K$). Formally $M = ||UF|| + (N - K)$. The domain of the metric is $\{0, 1, \ldots, N\}$.

Lemmas 2 through 6 facilitate the proof. Lemma 2 states that the metric is equivalent to N minus the number of known-false and irrelevant time steps. Lemma 3 states that metric M is a decreasing function. Lemma 4 states that if the set of unknown-if-false time steps contains two or more elements, and if the termination condition fails at the current time step in s, then the metric must eventually decrease. Lemmas 5 and 6 lie at the heart of the overall proof of correctness of *Soln1* as well as of later refinements. Lemma 5 demonstrates that if there exists an unknown-if-false time step at which C holds, then variable s will eventually be set to a time step at which C holds. This lemma will be reproven for each of the subsequent refinements, because the refinements are more sophisticated methods of exploring the interval $[1, K]$ via assignments to variable s. Lemma 6 implies that if C holds at a committed time step, then C will eventually hold at an unknown-if-false time step.

Lemma 2 $M = N - ||KF||$

Proof: By (td4), $||UF|| = K - ||KF||$. Substituting this expression into the definition of M completes the proof. □

Lemma 3 $\langle\ \forall m :: M = m$ unless $M < m\ \rangle$

Proof:

$\langle\ \forall m :: ||KF|| = m\ \textit{unless}\ ||KF|| > m\ \rangle$
  , by (td6)

$\langle\ \forall m :: N - ||KF|| = m\ \textit{unless}\ N - ||KF|| < m\ \rangle$
  , by last deduction, because N is constant

$\langle\ \forall m :: M = m\ \textit{unless}\ M < m\ \rangle$
  , substitute Lemma 2 into last deduction □

Lemma 4 $\langle\ \forall m :: M = m \land \neg C(s) \land ||UF|| > 1 \mapsto M < m\ \rangle$

Proof:

$\langle\ \forall t :: ||KF|| = t\ \textit{unless}\ ||KF|| > t\ \rangle$
  , by (td6)

$\langle\ \forall t :: \neg C(s) \land ||UF|| > 1 \land ||KF|| = t \mapsto ||KF|| > t\ \rangle$
  , Progress-safety-progress (Theorem (2)) applied to last deduction and (td5)

$\langle\ \forall m, t :: N - ||KF|| = m \land \neg C(s) \land ||UF|| > 1 \land ||KF|| = t \mapsto (||KF|| > t \land N - ||KF|| = m) \lor ||KF|| > t\ \rangle$
  , Progress-safety-progress (Theorem (2)) applied to last deduction and result of substituting Lemma 2 into Lemma 3

$\langle\ \forall m :: M = m \land \neg C(s) \land ||UF|| > 1 \mapsto M < m\ \rangle$
  , $||KF|| > t \land N - ||KF|| = m \Rightarrow \textit{false}$ in last deduction, because N is constant

□

Lemma 5 $C(s)$ *detects* $\langle\ \exists k : k \in UF :: C(k)\ \rangle$

Proof: Let $p = \langle\ \exists k : k \in UF :: C(k)\ \rangle$. From the definition of *detects* , we must show two things. First, $C(s) \Rightarrow p$; this follows from (td3) and (td4). Second, $p \mapsto C(s)$; this is proven below.

Observe that, by General Disjunction (Theorem (4)),

$$\langle \forall m :: \frac{\begin{array}{c} p \wedge [\|UF\| + N - K \leq 0] \mapsto \textit{false} \\ p \wedge [\|UF\| + N - K = 1] \mapsto C(s) \\ p \wedge [\|UF\| + N - K > 1] \wedge C(s) \mapsto C(s) \\ p \wedge [\|UF\| + N - K = m] \wedge m > 1 \wedge \neg C(s) \mapsto p \wedge [\|UF\| + N - K < m] \end{array}}{p \wedge M = m \mapsto (p \wedge M < m) \vee C(s)} \rangle.$$

Applying the Induction Theorem (6) to the conclusion yields the desired result, $p \mapsto C(s)$. Therefore we are left with establishing each of the four formulas in the hypothesis. Note that these four formulas are mutually exclusive and exhaustive assertions about the state of the simulation when p holds, based on the value of metric M.

Proof that $p \wedge [\|UF\| + N - K \leq 0] \mapsto \textit{false}$:

$p \Rightarrow \|UF\| \geq 1$
, predicate calculus

invariant $N - K \geq 0$
, by (Sim3)

$p \Rightarrow \|UF\| + N - K \geq 1$
, Substitution axiom applied to last two deductions

$p \wedge [\|UF\| + N - K \leq 0] \Rightarrow \textit{false}$
, predicate calculus on last deduction

$p \wedge [\|UF\| + N - K \leq 0] \mapsto \textit{false}$
, Theorem (1) applied to last deduction

Proof that $p \wedge [\|UF\| + N - K = 1] \mapsto C(s)$:

$p \Rightarrow \|UF\| \geq 1$
, predicate calculus

$p \wedge [\|UF\| + N - K = 1] \Rightarrow p \wedge \|UF\| = 1$
, by last deduction

$p \wedge [\|UF\| + N - K = 1] \Rightarrow p \wedge \|UF\| = 1 \wedge s \in UF$
, Substitution axiom on last deduction and (td3)

$p \wedge [\|UF\| + N - K = 1] \Rightarrow C(s)$
, simplify last deduction, and apply Theorem (1)

Proof that $p \wedge [\|UF\| + N - K > 1] \wedge C(s) \mapsto C(s)$**:** By predicate calculus, $p \wedge [\|UF\| + N - K > 1] \wedge C(s) \Rightarrow C(s)$; applying Theorem (1) completes the proof.

Proof that $p \wedge [\|UF\| + N - K = m] \wedge m > 1 \wedge \neg C(s) \mapsto p \wedge [\|UF\| + N - K < m]$**:**

(a) $\langle \forall m :: p \wedge M = m \wedge m > 1 \wedge \neg C(s) \Rightarrow \|UF\| > 1 \rangle$
, by (td3)

(b)
$\langle \forall m :: p \wedge M = m \wedge m > 1 \wedge \neg C(s) \Rightarrow p \wedge M = m \wedge m > 1 \wedge \neg C(s) \wedge \|UF\| > 1 \rangle$
, by previous deduction

(c) $\langle \forall m :: p \wedge M = m \wedge \neg C(s) \wedge \|UF\| > 1 \mapsto p \wedge M < m \rangle$
, apply Theorem (3) to Lemma 1 and Lemma 4

(d) $\|UF\| > 1 \Rightarrow M > 1$
, by (Sim3) and definition of M

(e) $\langle \forall m :: p \wedge M = m \wedge m > 1 \wedge \neg C(s) \wedge \|UF\| > 1 \mapsto p \wedge M < m \rangle$
, substitute (d) into (c)

(f) $\langle \forall m :: p \wedge M = m \wedge m > 1 \wedge \neg C(s) \mapsto p \wedge M < m \rangle$
, combine (b) and (e) □

Lemma 6 $\langle \exists k : k \in UF :: C(k) \rangle$ detects $\langle \exists k : 1 \le k \le K :: C(k) \rangle$

Proof: Let $p = \langle \exists k : k \in UF :: C(k) \rangle$ and $q = \langle \exists k : 1 \le k \le K :: C(k) \rangle$. We must prove two things. First, $p \Rightarrow q$; this follows from (td4) and the fact that $\neg C(0)$ holds by definition. Second, $q \mapsto p$; this follows from (td4) and (td7). □

Proof of PS1: Follows from (td1), Lemmas 5 and 6, and two applications of Theorem (7). □

Proof of PS2:

TCD *detects* $\langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ C(k)\ \rangle$
, by PS1

$TCD \Rightarrow \langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ C(k)\ \rangle$
, last deduction and definition of *detects*

$TCD \Rightarrow \langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ om = om(k)\ \rangle$
, (td2), (td3), and (td4)

$TCD \Rightarrow \langle\ \exists k\ :\ 1 \leq k \leq K\ ::\ om = om(k)\ \wedge\ C(k)\ \rangle$
, Combine last two deductions □

Proof of PS3: By the UNITY union theorem,[4, pp. 155-6]

$$\textbf{invariant}\ p \text{ in } P \wedge \textbf{invariant}\ p \text{ in } Q \Rightarrow \textbf{invariant}\ p \text{ in } P \square Q$$

Therefore PS3 follows from (Sim7) and (td8). □

Derivation of a program from the solution strategy specification: Figure 2 illustrates one possible program, *TD*, that meets specification *Soln1*. A proof that the Program *Soln1* meets specification *Soln1* is straightforward and is omitted in the interest of space. Informally, the *always* relationship embodies (td4); the assignment statement embodies the remaining the remaining assertions in *Soln1* and does not violate any assertion in specification *Sim*.

4.1.3 Refinement: Eliminating Congruent Time Instances

Informal description: The following refinement can further reduce the number of evaluations of C required to detect termination. Given a simulation model and a termination condition, it is often possible to infer that at certain time steps no model attribute has changed that will cause a change in the value of the termination condition. For example, in condition T2 of Section 1, the termination condition need not be evaluated at any time step at which a collision does not occur. Two time steps are *congruent*, denoted $j \cong j'$, for any time steps j and j', if $C(j) = C(j')$

```
Program TD

declare
  UF, KF: set of integers, s, OldK: integer
initially
  s=0 || KF=∅ || OldK=0

assign

    s := some element in UF     if ¬C(s) ∧ ||UF|| > 1 ∧ ¬TCD
    || KF := KF ∪ {s}           if ¬C(s) ∧ ||UF|| > 1 ∧ ¬TCD
    || UF:= UF- { s }           if ¬C(s) ∧ ||UF|| > 1 ∧ ¬TCD
    || om := om(s)              if C(s) ∧ ¬TCD
    || TCD:= true               if C(s) ∧ ¬TCD

  □⟨ ⟨ || j : OldK < j ≤ K :: UF:=UF∪{j} ⟩
    || OldK:=K ⟩

end { TD }
```

Figure 2: Program implementing specification *Soln1.*

is known to hold before the simulation executes. Congruence is a reflexive, symmetric, and transitive relation.

In the previous specification, s is assigned values from set *UF*, the set of all committed time steps at which the termination condition has not yet been evaluated. *Soln2* modifies this statement so that whenever the committed time threshold advances, newly committed time steps congruent to a time step in unknown-if-false are added to an additional set, *I*, which contains irrelevant time steps, to reduce the number of times that the termination condition is evaluated. Set *I* is a local variable to program *TD*.

One form of this refinement is employed by existing termination detection algorithms (Abrams and Richardson [1] as well as Lin [6]). In these algorithms, the agent responsible for detecting termination is only sent the values of attributes that are required to evaluate the termination condition when the values change.

Formal specification of solution strategy: The refined specification, *Soln2*, consists of replacing (td4)by (td9) through (td12). Initially, $I = \emptyset$.

Specification *Soln2*: [(td1)–(td3), (td5)–(td12)]

(td9) **invariant** $UF \cup KF \cup I = \{0, 1, 2, \ldots, K\} \wedge UF \cap KF \cap I = \emptyset$

(td10) **invariant** no two distinct elements of UF are congruent

(td11) **stable** $\langle\ \forall k\ :\ 0 \leq k \leq N\ ::\ k \in I\ \rangle$

(td12) **invariant** $\langle\ \forall k\ :\ k \in I\ ::\ \langle \exists k'\ :\ k' \in UF \cup KF\ ::\ k \cong k' \rangle\ \rangle$

The new assertions state the following. The set of committed time steps are partitioned into sets *UF*, of unknown-if-false time steps, *KF*, of unknown-if-false time steps, and *I*, of irrelevant time steps (td9). (The assertion that (td9) replaces partitions committed time steps into only two sets: UF and KF.) No two distinct elements of known-false are congruent (td10). (Assertions (td3) and (td10) imply that when the value of s is added to set KF, then s is assigned another unknown-if-false time step.) Time steps are never removed from the set of irrelevant time steps (td11). Each time step in the set of irrelevant time steps is congruent to some committed time step that is either unknown-if-false or known-false (td12).

Correctness proof of solution strategy: Variables UF and KF in the specification *Soln2* refer to different sets than in specification *Soln1*. Therefore, rather than prove that specification *Soln2* implies the assertion removed from *Soln1* ((td4)), we again demonstrate that that the problem specification (assertions PS1 through PS3) are met by any program satisfying *Soln2*.

The proof is similar to that for specification *Soln1*, but with the following modifications. First, replace each occurrence of $||KF||$ by $||KF \cup I||$. Second, modify the proof of Lemma 6 as explained below. (The proofs of Lemmas 2 through 5 as well as PS1 through PS3 do not depend on (td4) and hence hold for *Soln2*.)

Modified proof of Lemma 6: The proof of $p \Rightarrow q$ does not require modification. However, $q \mapsto p$ follows from (td9), (td7), and $\langle \ \forall k \ : \ k \in I \ :: \ \neg C(k) \ \rangle$. The last assertion follows because each element in I is congruent to some element in either UF or KF by (td12). □

Derivation of a program from the solution strategy specification: A program *TD* satisfying *Soln2* is the obtained by modifying the program in Figure 2 as follows. Add "I= ∅" to the **initially** section. In the **assign** section, modify the assignment to UF as follows:

```
□⟨ ⟨ || j : OldK < j ≤ K :: I:=I∪{j}      if ⟨∃ j' : j'∈KF∪UF:: j≅j' ⟩ ⟩
     ⟨ || j : OldK < j ≤ K :: UF:=UF∪{j}  if ⟨∀ j' : j'∈KF∪UF:: j≇j' ⟩ ⟩
   || OldK:=K ⟩
```

4.2 Reducing Termination Time through Reduced Evaluation

Informal description: Solution strategy *Soln1* requires at worst N evaluations of the termination condition to detect termination. This results from an exhaustive search of time steps in the interval $[1, N]$. Strategy *Soln2* can reduce the worst case number of evaluations to the number of equivalence classes of time steps, provided that a simulation program can efficiently identify congruent time steps. The next strategy, *Soln3*, improves the average case performance by exploring the interval $[1, N]$ in a sequence tailored to a particular termination condition. The sequence is prescribed by an *ordering function*, f, whose domain is a time step and the values of all simulation model attributes at that time step, and whose range is a positive integer time step. For convenience, we write $f(k)$, for some time step k, even though f may also be a function of simulation model attributes. An efficient ordering function could on the average detect termination sooner than strategies *Soln1* and *Soln2*, which prescribe no order. In general, in the worst case *Soln3* requires the same number of evaluations as *Soln2*. However, in the case of a stable termination condition, an appropriate ordering function can also reduce the worst case performance to as few as one.

Let $f^i(k)$ denote the i-fold composition of function f. Function f must satisfy the following properties:

$$f^i(0) \in \{1, 2, \ldots, N\} \text{ for } i = 1, 2, \ldots, N$$

$$f^{N+1}(0) > N$$

$$f^i(0) \neq f^j(0) \text{ for } i \neq j$$

The first property requires that the first N evaluations of function f map a time step to one of the first N time steps. The second property states that after function f returns the first N time steps, it must return a time step that the simulation will never compute. Finally, the images of function f applied to any two distinct time steps are distinct.

This solution strategy views the termination detection problem as one of searching the interval $[1, N]$ for a time step that satisfies the termination condition.

The solution strategy again requires variable s, denoting a time stamp. Again, s is initially zero. Whenever s contains a committed time step and the termination condition evaluates to *false* for time step s, s is assigned $f(s)$. Unlike *Soln1* and *Soln2*, variable s may assume uncommitted as well as committed time steps.

Examples of ordering functions: Ordering functions must be tailored to individual classes of termination conditions. Some examples are given below.

Exhaustive termination: The ordering function $f(k) = k + 1$ may be used with any termination condition. The Exhaustive Termination Algorithm of Abrams and Richardson[1], as well as the non-stable termination condition given by Lin[6] are both refinements of strategy *Soln3* with this ordering function.

Interval termination: Stable termination conditions permit a simple ordering function that can dramatically reduce the number of evaluations of the termination condition. If C holds at time step k, then it hold at all time steps larger than k. If the termination condition fails at some time step k, then simply wait c time steps, for any finite integer c, before reevaluating C, provided that $k + c \leq N$. Formally, $f(k) = k + c$.

In fact, selecting $c = N$ guarantees that the termination condition is evaluated exactly once, which is optimal in the number of evaluations. However, this may not minimize the wall clock time required to execute detect termination, as discussed later. The Interval Termination Algorithm of Abrams and Richardson is a refinement of *Soln3* with the ordering function $f(k) = k + c$.[1]

Predictive termination: A variety of predictive guesses may be formulated based on the exact form of the termination condition. One prediction is based on extrapolation from the points $(0,0)$ and $(s, c(s))$. For example, condition T2 from Section 1 can be written as $C(s) \equiv (c(s) \geq M)$, where $c(s)$ is the number of collisions that have occurred at the time step s. Extrapolation leads to the ordering function $f(s) = (Ms)/c(s)$. For example, if $M = 1000$ and at the 5-th time step 50 collisions have occurred ($c(5) = 50$), then $f(5) = 100$ guesses that 1000 collisions will have occurred by the 100-th time step.

A termination condition of the form $c(s) \in [a, b)$ can be handled by a method that is analogous to solving for roots of equations. One strategy is to find time steps a', b' such that $c(a') < a$ and $c(b') > b$, and then use binary search to locate a time step between a' and b' satisfying the termination condition. In this case $f(a', b') = \frac{b'-a'}{2}$. For example, consider condition T4 from Section 1, with $M = 1000$. Letting $c(s)$ again denote the number of collisions that have occurred, if $c(50) = 510$, $s = 100$, and $c(100) = 1903$, then the termination condition is next evaluated at time step 75.

More sophisticated predictions may be made; for example one could fit a curve to periodic observations of $(s, c(s))$ during simulation to obtain a better estimate of when $c(s)$ will equal M. A termination condition of the form $c_1(s) > c_2(s)$ can be handled by fitting curves to observations of functions c_1 and c_2, and then analytically solving for the intersection point.

An ordering function can be characterized as conservative or aggressive, based on the magnitude of $f(k) - k - 1$, which is the number of time steps that are skipped in successive evaluations of the termination condition. The earlier example of exhaustive termination ($f(k) = k + 1$) uses a conservative function that skips zero time steps. In contrast, the interval termination function skips $c-1$ time steps. The magnitude of c determines the degree of aggression. There is a trade-off between aggression and conservativeness: A conservative ordering function maximizes the number of evaluations of the termination condition, but will minimize the number of time steps that the underlying simulation computes. However, increasing the aggressiveness of the ordering function reduces the number of evaluations of the termination condition but will select a larger time step as the termination time. The relative cost of evaluating the termination condition and simulating another time step determines the number of time steps to skip that is optimal in the sense of minimizing the wall clock time required to execute a simulation and detect termination. The optimum is, unfortunately, not known before simulation.

Formal specification of solution strategy: Initially, $s = 0$.

Specification *Soln3*: [(td13)–(td16)]

(td13) *TCD detects* $C(s)$

(td14) **invariant** $TCD \Rightarrow om = om(s)$

(td15) $\langle\ \forall k :: s = k \wedge s \leq K \wedge \neg C(s) \mapsto k \in KF\ \rangle$

(td15) **invariant** $TCD \Rightarrow FP$

Specification *Soln3* states the same properties as (td1), (td2), and (td8) of *Soln1*. In addition, *Soln3* states that if the termination condition is *false* at committed time step s then s will be assigned $f(s)$ (td14).

Correctness proof of solution strategy: Because solution *Soln3* is fairly simple, we only outline the proof that the three assertions (PS1 through PS3) comprising the problem specification are met by any solution strategy that satisfies conditions (td12) through (td15) of *Soln1*.

Assertion PS1 can be proved by induction (Theorem 6) and (Sim2) using as the metric N minus the number of times that f has been evaluated so far. The metric is bounded below by zero. Assertion PS2 follows from (td12) and (td13). The proof of Assertion PS3 is identical to its proof in Section 4.1.2.

5 Mapping General Solution to Parallel Simulation Protocols

This section describes mapping termination detectors *Soln1* and *Soln3* of Section 4 to various parallel simulation protocols and parallel and distributed architectures. We omit *Soln1*, because *Soln2* is its refinement. Our mapping of *Soln2* to a parallel processor uses different processors to evaluate $C(k)$ for different values of k. In our mapping of *Soln3*, a single evaluation of $C(k)$ for a particular value of k can be mapped to a set of processors when possible.

The key decisions that must be specified are

1. how the multiple assignment statements of *Soln2* and *Soln3* are mapped to processes, and
2. how to reduce the worst case memory requirement of $O(N)$ in *Soln2* and *Soln3* so that the algorithm can run, at best, in no more memory than the underlying simulation requires.

Decision 2 arises because the last section assumed that the underlying simulation retains the final values of *all* attributes at *all* committed time steps. Transforming *Soln2* and *Soln3* to reduce the memory required will constrain the order in which the termination condition must be evaluated.

5.1 Mapping to Architectures

We consider three architectures. The first is a parallel asynchronous shared memory architecture, in which all processors can read and write any memory location, and each processor asynchronously executes a separate list of statements. The second is a distributed architecture, in which each processor again asynchronously executes a separate list of statements and has a local memory that it can read and write; processors communicate through channels. The third is a parallel synchronous shared memory architecture, in which all processors perform an operation upon reception of a common clock pulse; this is an SIMD architecture.

5.1.1 Mapping *Soln2* to Architectures

Asynchronous shared memory architecture: Each of L processors runs a copy of the termination detection program, *TD* (Figure 2, as modified for *Soln2*). The value of L must be chosen to balance the overhead of the *TD* against the time penalty of the *TD* over the underlying simulation. Each processor i has a private copy of variables s and *TCD*, denoted s_i and TCD_i, respectively. Sets *UF*, *KF*, and I are partitioned into disjoint sets (e.g., UF_i, KF_i, and I_i), with each processor being assigned one partition. Whenever additional time steps commit, the newly committed times are partitioned among the processors to each partition of sets *UF* and I. Therefore each processor i asynchronously transfers potential stopping time steps from set UF_i to s_i, evaluates $C(s_i)$, and either adds s_i to set KF_i or sets variable TCD_i to cause termination. For efficiency, each termination detection program may only be scheduled when there are a sufficiently large number of newly committed time steps, to reduce the overhead.

This solution can be expressed by modifying *Soln2* to quantify all assertions involving sets *UF*, *KF*, and I over all processors. In addition, assertions (td1) must be modified to detect the conjunct $C(s_1) \wedge C(s_2) \wedge C(s_L)$. Finally, it could be the case that multiple processors detect termination by the time that variable TCD is assigned *true*. This case is particularly likely for a stable condition. Therefore assertion (td2) is modified to select one of the detected time steps as the termination time at which output measures are reported.

Copies of the *TD* program can read the attributes required to evaluate the termination condition and calculate the output measure from shared memory.

Distributed architecture: Again, a copy of *TD* executes on a set of L processors. In addition, a central coordinator process is sent updates of the committed time horizon through a channel from the underlying simulation. All processes comprising the underlying simulation send the coordinator messages containing the values of attributes required to evaluate the termination condition for each newly committed time step. The coordinator then partitions the time steps and distributes them along with the appropriate attributes to the copies of the *TD* program via channels. Whenever a *TD* program evaluates the termination condition to *true*, it sends the time step to the coordinator via a channel. When the coordinator receives the first time step back from a *TD* program, it uses the time step as the termination time. The coordinator can then query processes of the underlying simulation to obtain attribute values required to evaluate the output measures at the termination time step. If there are many output measures, the evaluation of some can be assigned by the coordinator to a process running the *TD* program.

5.1.2 Mapping *Soln3* to Architectures

Single processor: The implementations of *Soln2* described above may or may not be efficient depending on the number of processors used, L, and the volume of attributes required for the termination condition. For example, in algorithms studied by Lin [6], the overhead in parallelizing the termination detection program outweighed the benefits of using multiple processors. Therefore we propose that termination detection program for *Soln3* be implemented on a single processor. The efficiency of *Soln3* lies in the fact that it reduces the number of evaluations of the termination required in the first place, rather than parallelizing many evaluations.

Synchronous shared memory architecture: One simple way to parallelize *Soln3* is to evaluate a termination condition at a single time step using multiple processors in an SIMD fashion. This technique can only be used for certain termination conditions, by exploiting properties of those classes. For example, decentralized consensus protocols [5] can be used to evaluate termination conditions whose operators form an Abelian group, and recursive doubling [10] can be used to evaluate conditions that are solutions to recurrence relations.

5.2 Mapping to Limited Memory

All solution strategies proposed provide few constraints on the order in which time steps are evaluated by the termination detection algorithm. *Soln2* tends to evaluate the condition in ascending time step order, but only because the committed time step threshold limits the size of set of unknown-if-false time steps. *Soln3* can use any possible sequence by an appropriate ordering function. In each of these solutions, memory requirements for termination can be minimized by constraining the order of evaluation.

Optimistic Protocols: Optimistic protocols have an inherent advantage over all other protocols for termination detection because they can always recompute any needed attribute at any simulation time step. For example, consider *Soln3* with a predictive ordering function that evaluates the condition first at time step N and then uses bisection to search for a time step at which the termination condition holds. This implies that the simulation needs to keep $O(Nn)$ storage, for N time steps and n attributes, under the assumption that every attribute is needed for evaluation of either the termination condition or some output measure. An optimistic protocol is nicely suited to this case, because it can recompute attributes for a termination detector that evaluates the termination condition in an arbitrary time step order. An optimistic protocol can computer termination in a minimal amount of memory. The use of an optimistic protocol is detailed in Abrams and Richardson[1].

Conservative-Synchronous: Conservative-synchronous protocols are attractive for termination detection because the termination condition can be evaluated at each global synchronization point with minimal synchronization overhead. The method can be augmented with a limited state saving and rollback method to obtain the same flexibility in regenerating attribute values for arbitrary simulation time steps as in optimistic protocols. The use of a conservative-synchronous protocol is detailed in Sanjeevan and Abrams[9].

Conservative-asynchronous: Conservative-asynchronous protocols are appear poorly suited for general termination conditions, because they lack the ability to regenerate attribute values at arbitrary time steps. In fact, the only way that these protocols can perform this function is by including the state saving, rollback, and anti-message functions that are present in optimistic protocols.

However, conservative-asynchronous protocols are well-suited to stable termination conditions. This is because if the termination detector finds that the termination condition is satisfied at time step k, then it can select a time $k' > k$ that is in the future of all processes in the underlying simulation as the termination time step. Time step k' can then be distributed to all underlying simulation processes, so that they stop when they reach time step k', and can then send attribute values at time step k' to the termination detector for calculation of output measures. This method is further explained in Abrams and Richardson[1].

6 Conclusions

For practitioners that just require batch simulation of a system for T simulation time units, complex simulation termination rules such as those listed in Section 1 are unnecessary. Therefore one could ask whether it is necessary to worry about complex termination rules.

We expect that in the future, as simulation more often is done with interactive, asynchronous user input; on a parallel or distributed architecture; and in more sophisticated settings, such as simulation-in-the-loop, complex termination rules may grow in importance.

In addition, the ability to implement complex termination rules in parallel simulations opens up the possibility of using simulation as a tool to search the transient behavior of a complex system. For example, one could run a simulation of a large telephone or communication network and make queries such as "when did congestion of certain switching nodes first develop" and "what was the system state at each time that the maximum utilization of any communication link in the system exceed 50%".

Given the need for solving the simulation termination problem, the paper presents two general and incompatible strategies for termination. The first, *Soln2*, is a novel algorithm that tries to minimize the cost of termination detection by evaluating many time steps in parallel. An advantage of the algorithm is that it is scalable; that is, the detector will not become a bottleneck as the size of the simulation model grows (this may preclude use of a single process as the termination detector, as Lin does [6]). The second solution, *Soln2*, is novel in its use of guesses of when the simulation will terminate. The guesses are embodied by an ordering function. A suitable ordering function subsumes termination algorithms proposed in the literature.

Our solutions suggest that a simulation modeler should use a stable termination condition and to minimize the number of attributes required for evaluation of the termination condition and all output measures. This permits use of optimistic, conservative-synchronous, or conservative-asynchronous simulation protocols. If the modeler cannot do this, then an optimistic protocol should be used for its flexibility to recompute whatever old attribute values are required. Conservative-synchronous protocols can also be used, if a limited ability to checkpoint and recompute states at synchronization points is added. Conservative-asynchronous protocols cannot be used, unless they are augmented with the same rollback and cancellation mechanism present in an optimistic protocol.

When a non-stable condition is used, a simulation modeler is also advised to devise a predictive ordering function and use termination strategy *Soln3* to search committed time steps in a sophisticated manner, for example by bisection, to minimize the number of evaluations of the condition. A suitable predictive function could make the cost of using a non-stable conditions as low as that of a stable condition.

A number of open problems on simulation termination exist:

- formulation of ordering function required by specification *Soln3*,
- formulation of algorithms to terminate time-quantified termination conditions,
- complexity analysis of the parallel simulation problem to establish fundamental limits on the performance of parallel simulation given certain difficult classes – such as non-stable and time-quantified – of termination conditions,
- further empirical studies measuring the cost of termination in actual simulations (two cases are reported by Lin [6] and Sanjeevan and Abrams [9]),
- further refinement of the general algorithms given here to obtain efficient implementations for particular simulation protocols and parallel computer architectures.

References

[1] M. Abrams and D. S. Richardson, "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation," *Proc. Advances in Parallel and Distributed Simulation,* Society for Computer Simulation, Anaheim, Jan. 1991, pp. 86-91.

[2] M. Abrams, E. Page, and R. Nance, "Linking Simulation Model Specification and Parallel Execution Through UNITY," *Proc. 1991 Winter Simulation Conference*, Phoenix, AZ, Dec. 1991.

[3] K. M. Chandy and R. Sherman, "Space-Time and Simulation," *Distributed Simulation 1989*, SCS, Jan. 1989, pp. 53-57.

[4] Chandy K. M. and J. Misra, *Parallel Program Design: A Foundation*, Reading, MA: Addison Wesley, 1988.

[5] T. V. Lakshman and V. K. Wei, *Efficient Decentralized Consensus Protocols Using Specially Structured Communication Graphs*, Tech. Memo TM-ARH-0011042, Bell Communications Research, Red Bank, N.J., 1988.

[6] Y. B. Lin, "On Terminating a Distributed Discrete Event Simulation," Bellcore, submitted for publication, 1991.

[7] Y. B. Lin and E. D. Lazowska, "Design Issues for Optimistic Distributed Discrete Event Simulation," submitted to *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[8] D. S. Richardson, *Terminating Parallel Discrete Event Simulations*, M.S. thesis, TR 91-9, Computer Science Department, Virginia Polytechnic Institute and State University, May 1991.

[9] V. Sanjeevan and M. Abrams, "The Cost of Terminating Synchronous Parallel Discrete-Event Simulations," *Proc. 1991 Winter Simulation Conference*, Phoenix, AZ, Dec. 1991.

[10] H. Stone, *High-Performance Computer Architectures*, 2nd ed., Reading, MA: Addison Wesley, 1990, pp. 232-233.