UNITY Algorithms for Detecting Stable and Non-Stable Termination Conditions in Time Warp Parallel Simulations

By D. Richardson and M. Abrams

TR 90-62

UNITY Algorithms for Detecting Stable and Non-Stable Termination Conditions in Time Warp Parallel Simulations

by D. Richardson and M. Abrams

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

December 19, 1990

ABSTRACT:

This paper extends work done by Abrams and Richardson on the topic of implementing global termination conditions and collecting output measures in parallel simulation. Concentrating on the Time Warp method for parallel simulation, an improved categorization scheme for termination conditions is presented, as well as algorithms written in UNITY notation to implement each category.

CR Categories and Subject Descriptors: I.6.0 [**Simulation Modeling**] : General; I.6.1 [**Simulation Modeling**] : Simulation Theory - *model classification, types of simulation* ; D.3.3 [**Programming Languages**] : Language Constructs - *concurrent programming structures* ; G.I.0 [**General**] : Parallel Algorithms

Additional Keywords and Phrases: parallel processing, distributed simulation.

TABLE OF CONTENTS:

APPENDICES:

1.0 INTRODUCTION

To date, research regarding parallel simulation has not been concerned with implementing arbitrary, global termination conditions and collecting the corresponding output measures. Local or "trivial" termination conditions have been used, such as terminating when each processor reaches a certain time, or terminating when each processor has processed a prespecified number of jobs. This paper proposes methods to accomplish termination of parallel simulations when it is impossible to develop a good local condition for termination.

Using current parallel simulation techniques it is not possible to implement every simulation model that can be implemented using sequential simulation. For example, suppose that a measurement is desired when two processors in the parallel simulation have processed a total of N jobs. Due to the asynchronous nature of parallel simulation, one processor will reach a certain point in simulation time before the other. Thus, once data from both processors has been collected for that time, and it has been determined that N messages have been processed, the processor which first reached that simulation time will have calculated beyond it.

Three key problems can be identified. Firstly, how can output measures be obtained from a time in the past of the parallel simulation? Secondly, how can each of the asynchronous processes compromising a simulation be brought to a halt once the termination time has been determined? Thirdly, how often should the termination condition be evaluated? The first two problems do not arise when a trivial termination condition is used because the prespecified termination point is designed for collecting the possible output measures. The third problem addresses efficiency; performance could suffer if the condition is evaluated too often, but evaluating too infrequently risks missing a time that satisfies the termination condition.

This paper categorizes global termination conditions, and presents algorithms to determine when such termination conditions have been accomplished, obtain output measures, and terminate the simulation. Efficiency is addressed for each category; known properties of a category are exploited to make the algorithm more efficient.

2.0 CATEGORIZATION

2.1 New Categorization Scheme

All termination conditions with a relatively short calculation time can be represented by their stability properties, or a combination of their component's stability properties.

A Backus-Naur form of representation is used to describe termination conditions. The terminal symbols ***f(t,.)*** and ***g(t,.)*** specify relationships among the values of a chosen set of simulation model attributes (".") at time t. W denotes any set of times.

<termination_condition>	::=	*<term>*	(1)
		| *<term>* ∧ *<termination_condition>*	(2)
		| *<term>* ∨ *<termination_condition>*	(3)
<term>	::=	***f(t,.)*** *<relation>* ***g(t,.)***	(4)
		| ∀t, t ∈ W, ***f(t,.)*** *<relation>* ***g(t,.)***	(5)
<relation>	::=	< | <= | <> | = | => | >	

2.2 Examples

Examples of *<term>* are as follows, with the number preceding each example referring to the corresponding statement from the BNF expressions above:

(4)	"the number of jobs processed at simulation time t is at least N"	***f(t,.)*** >= N
(4)	"the number of jobs processed at simulation time t is exactly N"	***f(t,.)*** = N
(5)	"for all simulation times less than t, the number of jobs processed is less than N"	∀i, ti < t, ***f(ti,.)*** < N

Examples of < *termination_condition* > are:

(1)	"the number of jobs processed at simulation time t is at least N"	***f(t,.)*** >= N
(2)	"the number of jobs processed at simulation time t is exactly N **and**	

the simulation time is at least M" $f(t,.) = N \wedge t >= M$

(3) "the number of jobs processed at simulation time t is exactly N **or** the simulation time is at least M" $f(t,.) = N \vee t >= M$

Addressing the stability of each of the above termination conditions: the first example presented to illustrate (1) is a stable condition *S*, for once it holds it will continue to hold; the second example for (1) is a non-stable condition *s;* the example for (2) consists of the conjunction of non-stable and a stable condition $s \wedge S$; the example for (3) consists of a non-stable condition unioned with a stable condition $s \vee S$. These are listed in Table 1:

termination category	*example*
S	$f(t,.) >= N$
s	$f(t,.) = N$
$s \wedge S$	$f(t,.) = N \wedge t >= M$
$s \vee S$	$f(t,.) = N \vee t >= M$

Table 1: Examples of termination conditions from various termination categories

Sections 5, 6, and 7 present algorithms to solve simulations with termination conditions that correspond to any of the above situations.

2.3 Illustration Using Space-Time Framework

Chandy and Sherman's space-time diagrams [1989] will be used to illustrate termination conditions. Space will be used to represent the *set of all attributes* in a simulation model.

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).

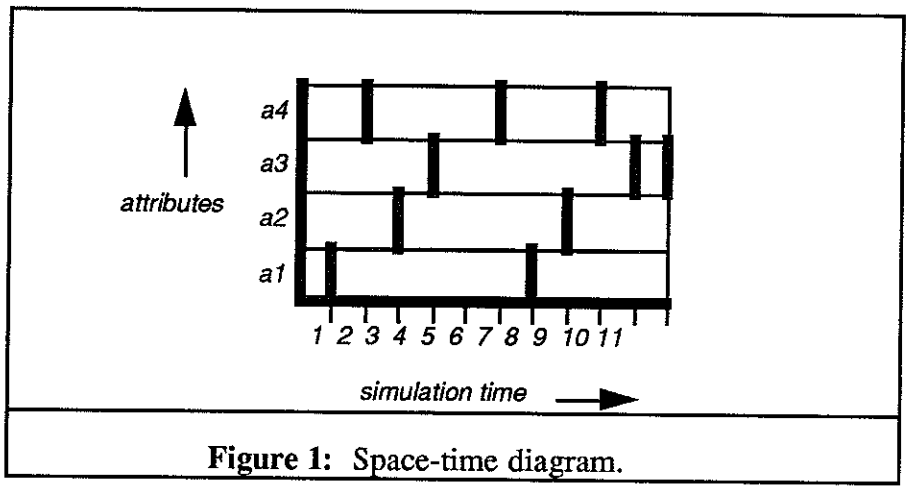

Figure 1: Space-time diagram.

A stable termination condition can be illustrated by a space-time diagram such as that in Figure 2a. The termination condition will be false until a certain simulation time; after that time, the termination condition will be true for all subsequent times. As mentioned above, an example of a stable termination condition is "at least N jobs have been processed".

A non-stable 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 non-stable termination conditions will only be true during one interval of time: an example would be the termination condition of "exactly N 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 ($f(t,.) = 3$)" is non-stable, because the telephones can always be in motion (Figure 2c).

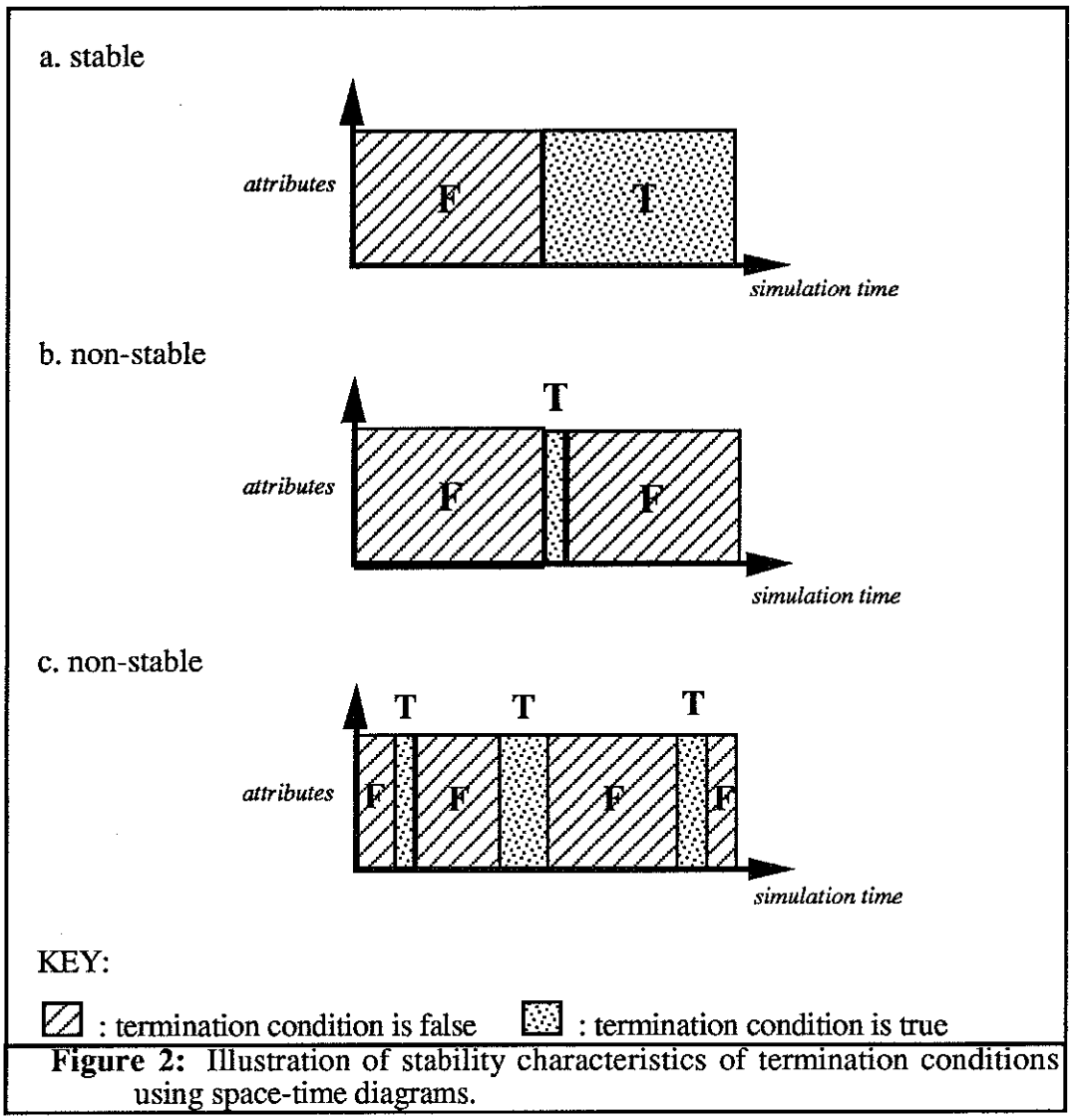

Figure 2: Illustration of stability characteristics of termination conditions using space-time diagrams.

3.0 TIME WARP ALGORITHM

3.1 Background Information

Before algorithms to implement various termination conditions can be presented, the underlying simulation that the termination method will terminate must be described. The parallel simulation method described here is the most utilized optimistic protocol, "time warp" [Jefferson 1985]. The termination algorithms are based on time warp because one of the conclusions made by Abrams and Richardson [1990] is that optimistic protocols are

better suited than conservative protocols for the additions necessary to implement a global termination condition*.

3.2 Specification Method

The time warp algorithm as well as the termination algorithms are specified using UNITY, a notation developed by Chandy and Misra [1987]. UNITY emphasizes a model of computation, a specification notation, and a theory for proving the correctness of specifications. The methods provided to compose larger programs from smaller ones is useful for our purpose because the termination algorithms will be merged with the underlying time warp algorithm. UNITY provides heuristics for refinement of programs which can be used to obtain 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 does not incorporate a method to terminate a program. A program is said to reach *fixed point* (FP) when all the values on the left and right sides of each assignment in the program are identical, so that it makes no difference whether the execution continues or terminates.

3.3 The Time Warp Algorithm

Program "*time_warp_process*" (see Appendix A) describes the time warp algorithm in UNITY. Briefly, the time warp simulation consists of a set of logical processes (LP's) communicating by message passing. Each LP will select the message with the lowest associated time, and execute the actions dictated by the message. If a message arrives that has a lower time than one already processed, "rollback" must occur. The LP will "undo" the actions that were done out of turn by erasing all changes in the state of the LP, and

* Functions such as calculating GVT, saving states, and collecting fossils to calculate output measures are mostly incorporated into time warp, but would have to be added to a conservative protocol.

sending antimessages to cancel out any messages sent to other LP's. The LP will begin forward processing again by restoring itself to a state that was saved before the time of the straggler message.

3.4 Graphical Illustration of Time Warp Algorithm Components

The components of a time warp process are illustrated in Figure 3.

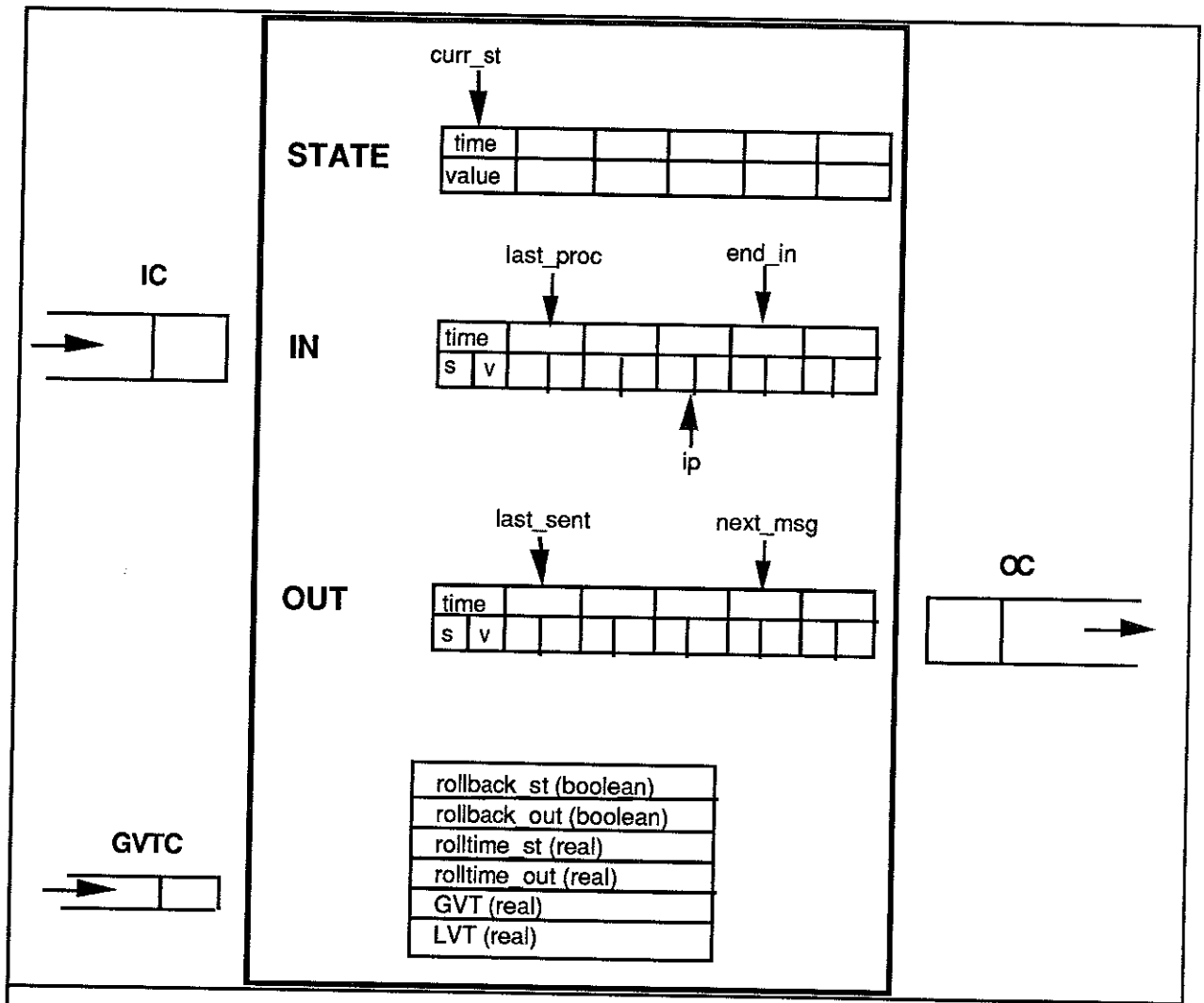

Figure 3: Graphic illustration of the components of a time warp process

Array **STATE** stores the values of model attributes that are local to the time warp process at various simulation times. Pointer **curr_st** indicates the next unused element of STATE to be filled. **IN** holds input messages received via input channel **IC** from other

processes , **end_in** indicates the next unused array element, **last_proc** points to the last message already processed by the process, and **ip** is used to order incoming messages by time. **OUT** holds all output messages produced by the process to be sent to other processes, **next_msg** indicates the next unused array element, and **last_sent** keeps tract of which messages have been transmitted over the output channel **OC**. **GVTC** is a channel that is read to obtain values of global virtual time (**GVT**) produced by a GVT algorithm. Variables **rollback_st, rollback_out, rolltime_st,** and **rolltime_out** are all used to manage rollbacks, which correct for messages processed out of timestamp order.

A group of processors each perform the time warp algorithm concurrently and asynchronously. The interconnection between the processors is not specified; the system can be loosely or tightly coupled. The time warp algorithm of Appendix A does not terminate or reach fixed point; each statement will be executed infinitely often.

4.0 DETAILS ON IMPLEMENTING A GLOBAL TERMINATION CONDITION

To implement a global termination condition and calculate the corresponding output measures, a new process must be introduced into the time warp simulation: the *termination detector*. In order for the termination detector to evaluate the termination condition for each arbitrary simulation time t, it must have access to the values of all required attributes at time t.

4.1 Centralized Solution

This paper assumes a centralized design for implementation of the termination detector. One process (represented by the algorithms *interval_termination, exhaustive_termination,* and *interval_than_exhaustive_termination* from Appendixes B, C, and D respectively) is assumed to collect all information needed to determine when the termination condition is satisfied, cause all participating time warp processes to terminate, and calculate the output measures. The solutions presented in this paper do not exclude the possibility of a distributed solution being developed; in fact, one of the strengths of the UNITY notation used to present the algorithms is that refinement of an algorithm to accommodate a particular architecture (such as a distributed system) is possible.

4.2 Methods for Collecting Output Measures

Abrams and Richardson (1990) propose the following three methods to calculate global output measures.

Dissociative termination requires that the termination detector be informed of the values of attributes needed to calculate the output measures at every t for which the termination condition is calculated. If the attributes needed to calculate the termination condition are different from the attributes needed to calculate the output measures, this method would involve additional data being transmitted to the termination detector, and only the data corresponding to the termination time would be used.

Retrospective termination requires the time warp processes to transmit the values of the attributes needed for output measure calculation only when a termination time t has been discovered by the termination detector. This requires an additional feedback loop to be incorporated into the system; feedback that indicates to the time warp simulation that the attribute values required for calculating output measures corresponding to time t need to be sent, and a response by the simulation processes of the appropriate values being sent to the termination detector. That value will be in the past of each process in the simulation, for the termination condition can only be calculated once all processes have passed that point in time. Fossil collection must be altered to allow for this.

Prospective termination will only work for stable termination conditions. Once the termination condition has been determined to be true, the termination detector will inform all time warp processes that they *will* terminate at a certain time t that is in the future of all simulation processes. At that point, each time warp process will send the attribute for that t to the termination detector, and terminate. This avoids the fossil collection difficulty of the *retrospective* method.

For the sake of simplicity of the algorithms developed in this paper, it is assumed that the values needed for calculating the output measures are the same as the values needed to calculate the termination condition* . Thus, *dissociative termination* is implied, for the termination detector has all the information necessary to calculate the output measures for every t. However, *retrospective termination* would not be difficult to implement: what must be additionally satisfied is the ability to send the needed attributes at a t requested by

* This is not an unreasonable assumption. For example, if the termination condition is "the total number of jobs processed is at least N" and the output measure desired is the average processing time of jobs, this would be valid.

the termination detector, and a method to delay fossil collection until it is known that attributes from the old times will not be requested.

5.0 STABLE TERMINATION CONDITIONS

The algorithm presented to implement a stable termination condition is named *interval_termination* because the stability guarantees that the termination condition need only be calculated in intervals rather than exhaustively. This idea is illustrated in Figure 4. 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.

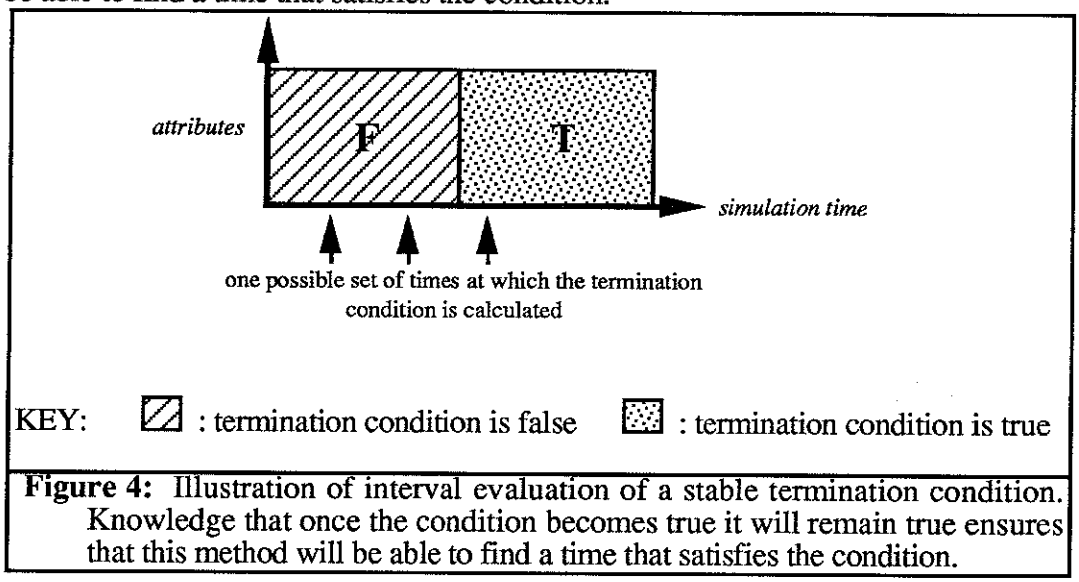

Figure 4: Illustration of interval evaluation of a stable termination condition. Knowledge that once the condition becomes true it will remain true ensures that this method will be able to find a time that satisfies the condition.

5.1 Justification of the Interval Termination Method

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 5. 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 timing 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.

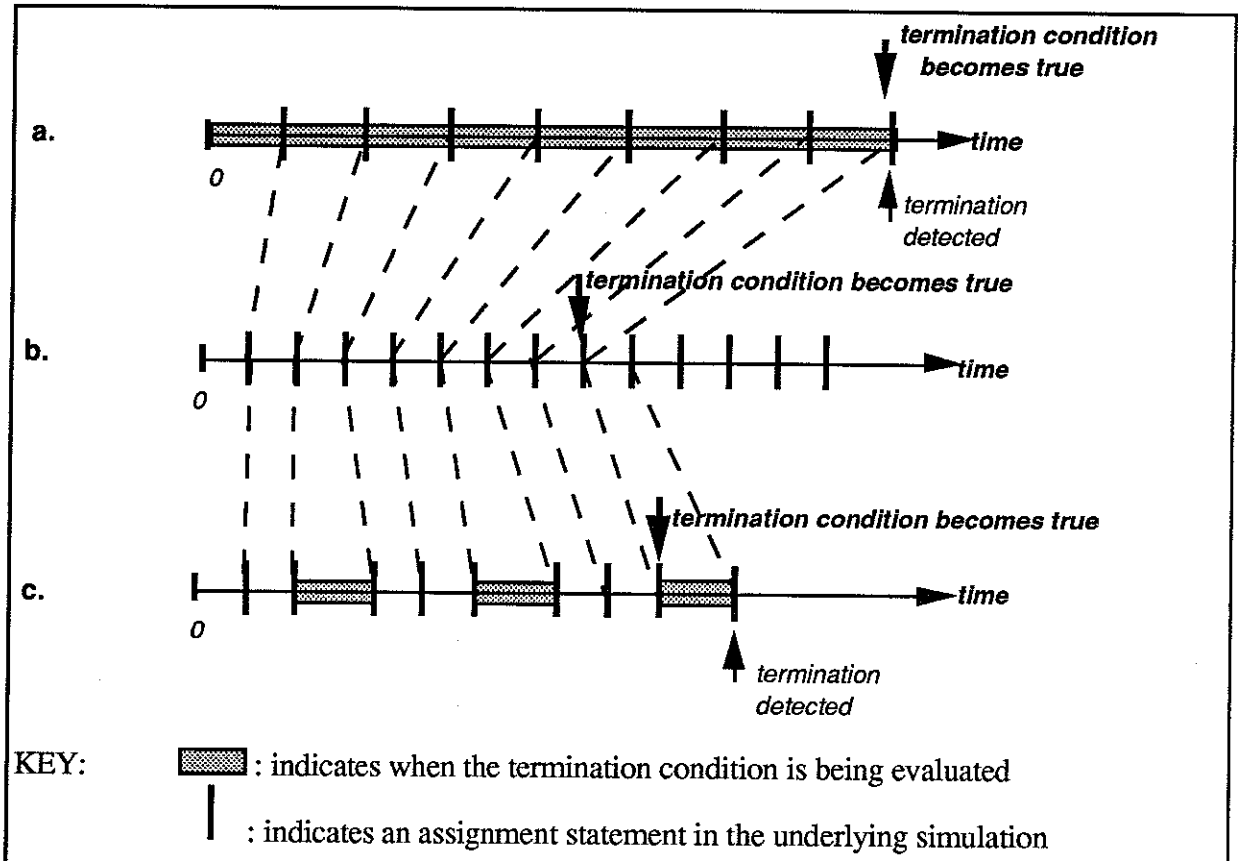

Figure 5: Justification of the interval termination method. Time line (b) represents the underlying simulation. The comparison between lines (a) and (c) shows that if the termination condition is evaluated exhaustively as in (a), termination will likely be detected at a later time than if it is evaluated over intervals as in (c).

5.2 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 each calculation, in determining the appropriate attributes to send to the termination detector, sending the attributes, saving the attributes until the condition can be calculated, etc. 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 noticed for 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 the simulation that a problem will most likely take at least five hours to compute, 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!

5.3 The Interval Termination Algorithm

Two UNITY programs are included in the Appendix B. The first program, *changes_to_tw_for_interval_termination,* specifies what additions need to be made to the time warp algorithm given in Appendix A so that processes running *time_warp_process* may transmit needed data to the termination detector, and be terminated (reach fixed point) when the termination condition is true. The termination detector runs the second program in Appendix B, *interval_termination.* This algorithm specifies what must be done to determine a) when the termination condition becomes true, b) collect the output measures, and c) shut down the simulation.

5.4 The UNITY Terminology of "superposition" and "union"

In UNITY terminology, *changes_to_tw_for_interval_termination* will be **superposed** onto *time_warp_process,* and *interval_termination* will be **unioned** with the result of the 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). This is useful, for example, to describe an application program and an operating system; the higher layer application program calls on the operating system routines, though the operating system never calls on an application program.

A unioning of two *component* programs to form a *composite* program is often used in UNITY to be able to express a complex problem. In the union of *interval_termination* and *time_warp_process,* the corresponding sections of the two programs are merged together to form the composite program (define section with define section, initially section with initially section, etc.). The union mechanism allows UNITY programs to follow from the software engineering principle that a large program should be composed from a number of smaller component programs, where each component is developed and understood by itself.

The superposed program *changes_to_tw_for_interval_termination* specifies only two additions to *time_warp_process*. The first is to add a channel from the time warp process to the termination detector to allow the attributes to be transmitted, the second is to add a variable "tw_terminate", which the termination detector can cause to become true once the termination condition becomes true, causing the time warp process to stop.

5.5 Graphical Illustration of Interval Termination Algorithm Components

The components of the termination detector running *interval_termination* are illustrated in Figure 6.

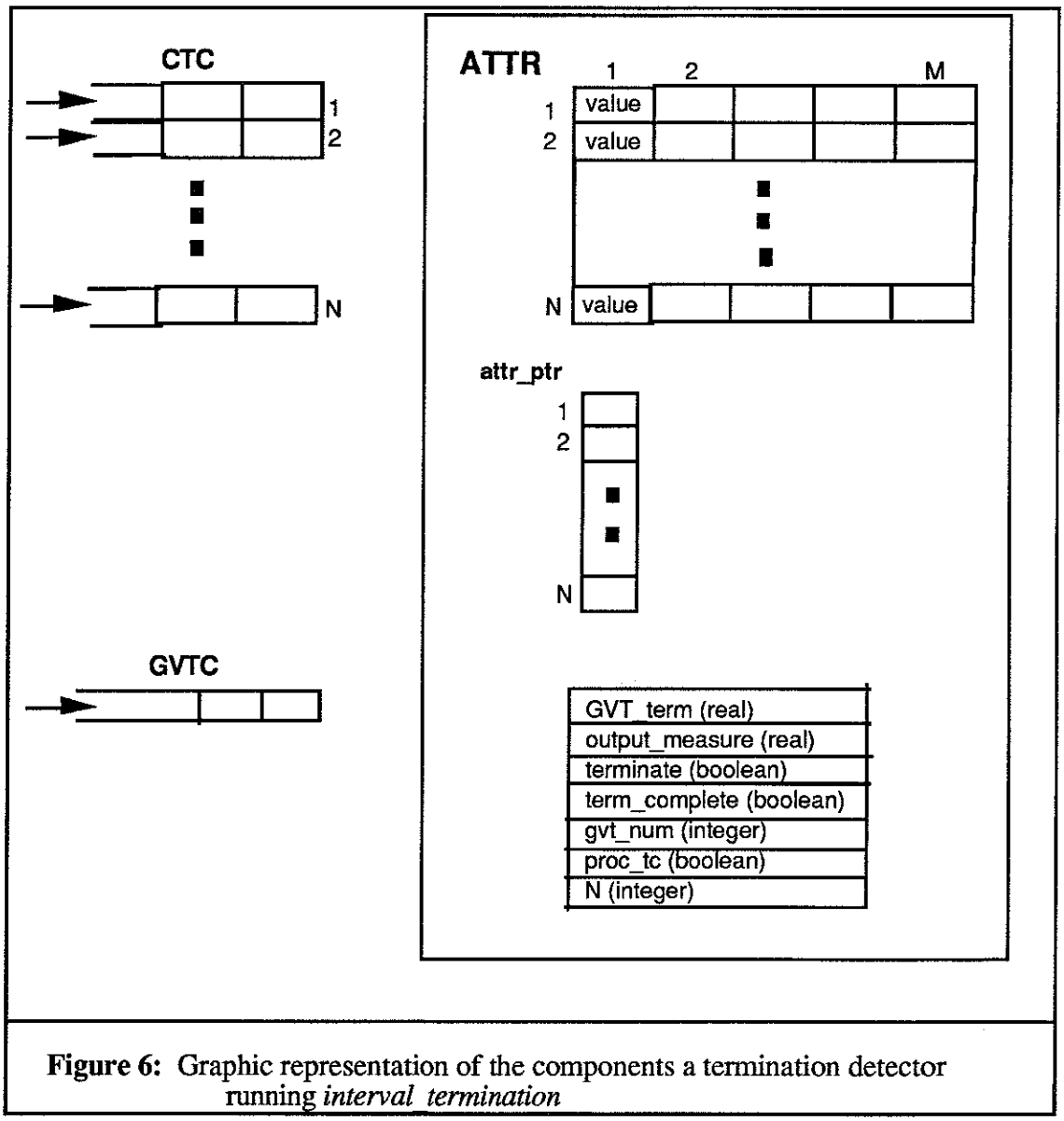

Figure 6: Graphic representation of the components a termination detector running *interval_termination*

An incoming channel from each of the **N** time warp process will deposit attribute data into array **ATTR**. Element *i* (1 <= *i* <= N) of array **attr_ptr** contains the largest *j* such that element *i,j* of **ATTR** contains an attribute value. The **GVTC** (channel for global virtual time updates) will update variable **GVT_term** in the termination detector as it updates GVT in all time warp processes; the number of updates received so far is recorded by **gvt_num**. **Proc_tc** indicates if the termination condition has been processed yet for the last GVT update received. **Terminate** indicates the results of the termination condition calculation; once it becomes true the **output_measure** is calculated. **Term_complete** insures that the output_measure is only calculated once, then the *interval_termination* process reaches a fixed point.

5.6 Assumptions and Simplifications

The main assumption made is that the topology of the time warp processes is static. It must be known beforehand that N processes are contributing attributes to the termination detector, and this number may not change. This requirement was made for the sake of simplicity, and will be maintained in all the termination algorithms given in this paper.

The second assumption is a small interval length. This algorithm allows calculation of the termination condition *every time GVT is updated.* This can be easily altered if desired; the addition of a single conditional could cause calculation after every N updates.

6.0 NON-STABLE TERMINATION CONDITIONS

The algorithm presented to implement a non-stable termination condition is entitled *exhaustive_termination*, because each time an attribute required to evaluate the termination condition changes, the termination condition must be reevaluated. The termination condition can not be calculated only over intervals, for a time that satisfies the termination condition may never be found. Take the example of the non-stable termination condition "exactly N jobs have been processed." This is represented with the space-time diagram of Figure 7. Also indicated in the figure are calculation points when the termination condition may be evaluated if an interval termination scheme was utilized. As can be seen by the illustration, the time when the attributes of the system cause the termination condition to be true does not happen to be one of the interval times where the condition is evaluated, thus termination could not occur.

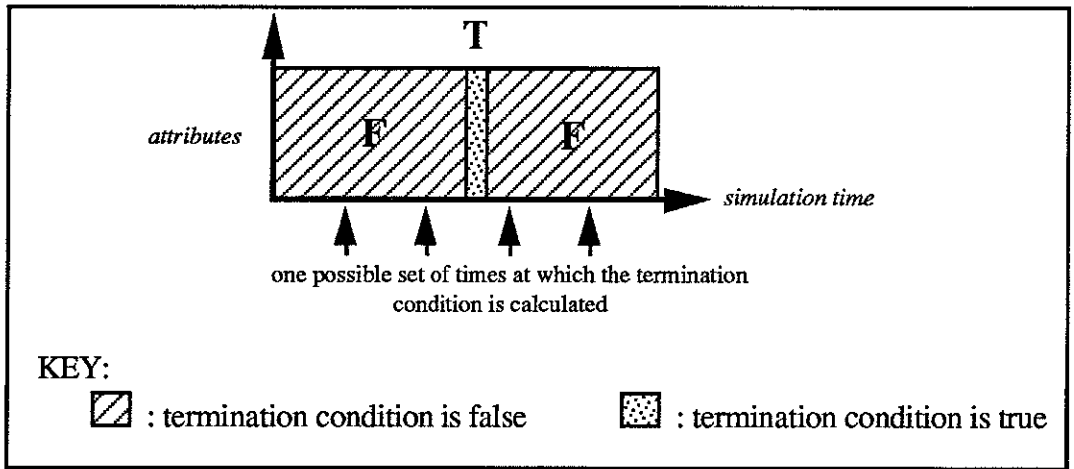

Figure 7: Illustration of interval calculation of a non-stable termination condition. The simulation time where the condition is true would not be tested by *interval_termination* using the indicated set of times, thus non-stable conditions can not be solved using this method.

For a non-stable termination condition, the condition must be recalculated *every time any attribute contributing to the condition changes.* Thus each time warp process must send updates to the termination detector whenever its attributes changes.

6.1 The Exhaustive Termination Algorithm

Two UNITY programs are included in the Appendix C to solve exhaustive termination of a parallel simulation using a non-stable termination condition. The first program, *changes_to_tw_for_exhaustive_termination,* is the superposed program that specifies what additions need to be made to the time warp algorithm given in Appendix A in order that processes running the time warp algorithm may contribute the data needed by the termination detector, and be terminated when the termination condition is true. The termination detector will be running the second program in Appendix C, *exhaustive_termination* which will be unioned with the composite *time_warp_process.* This algorithm specifies what must be done to determine a) when the termination condition becomes true, b) collect the output measures, and c) shut down the simulation.

Only three changes need to be made to the time warp process. The first is to add a channel (**CTC**) from the time warp process to the termination detector to allow the attributes to be transmitted. The second is to add variable **tw_terminate**, which the termination detector sets true once the termination condition becomes true, causing the time warp process to stop. The third change required (the only one different from the changes

required by the *interval_termination* algorithm) is an additional indexing variable **send_ptr** that keeps tract of which states have already been sent to the termination detector. This was not needed for *interval_termination* because that algorithm simply sent one state every GVT update interval. *Exhaustive_termination* must send every state.

The components of the termination detector running *exhaustive_termination* are identical to those illustrated in Figure 6 with the exception of additional variable **prev_calc_time** that stores the time when the termination condition was last calculated. This variable is needed because the condition must be evaluated every time any attribute in the system changes rather than once every GVT update.

6.2 Assumptions and Simplifications

As in the *interval_termination* algorithm, the main assumption made is that the topology of the time warp processes is static. It must be known beforehand that N processes are contributing attributes to the termination detector, and this number may not change.

The algorithm is simplified by sending all the states of the time warp processes to the termination detector. In reality, updates only need to be sent when a needed attribute has actually changed (many states will contain the same value for the needed attribute). This overkill was allowed to avoid detailing an identity detection function.

7.0 CONJUNCTIVE TERMINATION CONDITIONS

In section 2.3, it was shown 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 τ". 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 two stable conditions, the conjunction of two non-stable conditions, and the conjunction of a stable and a non-stable condition.

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 two stable conditions. Once the two stable conditions become true they will remain true: thus the conjunction of two stable conditions is a stable condition. The conjunction of two non-stable conditions is similarly a non-stable condition, thus must be solved using the exhaustive method (see Table 2 on page 22).

7.1 The Conjunction of a Stable and a Non-Stable Condition

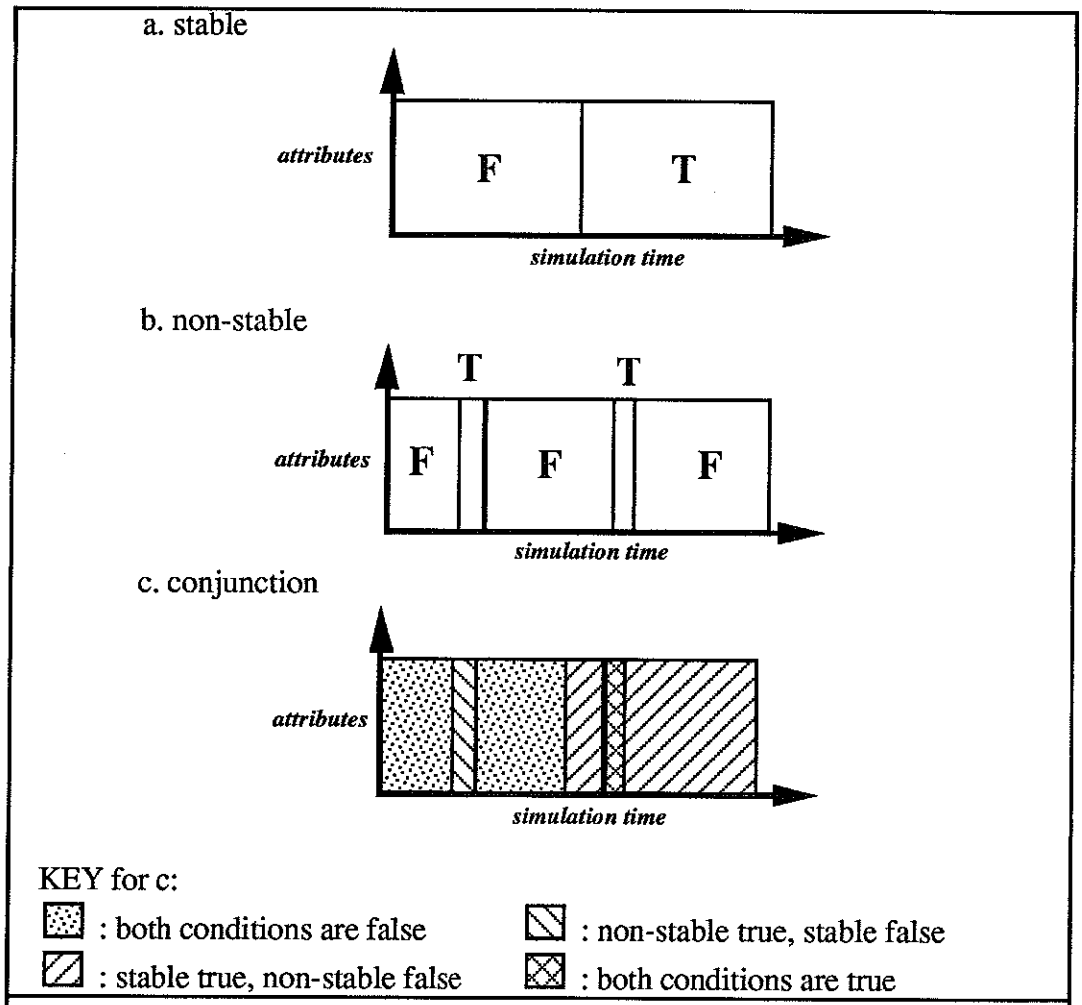

Figure 8: (c) is an illustration of the conjunction between a stable condition (a), a non-stable condition (b). Only when both conditions are true simultaneously can termination occur.

Figure 8 illustrates this combination. Figure 8a illustrates a stable property, 8b a non-stable property, and 8c represents the conjunction of the two. Only when both

properties are true concurrently will the overall termination condition be satisfied and termination of the simulation can occur.

The conjunction of a stable and a non-stable condition 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 will begin by being false, and will remain false until it becomes true and remains true, it seems that any exhaustive calculation before the stable condition becomes true will be unproductive.

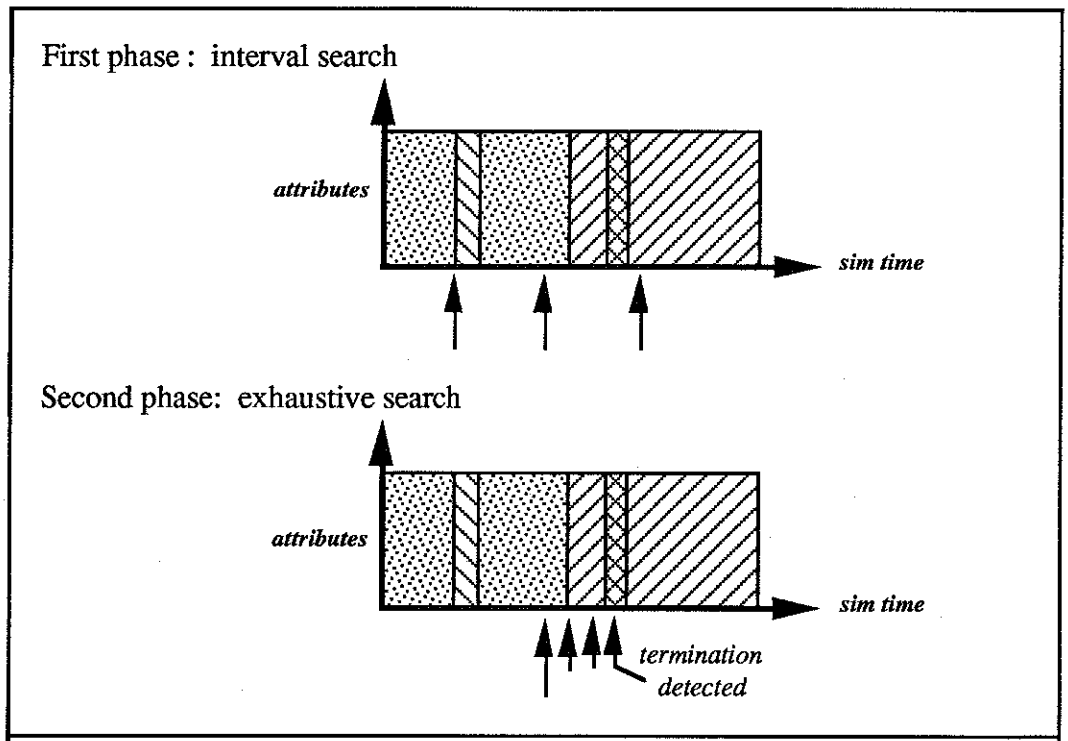

Figure 9: Illustration of the interval_then_exhaustive method to detect termination when the termination condition is a conjunction of stable and non-stable conditions. In the first phase, the interval termination method is used, until the first test where the stable condition is true is found. The second 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.

The proposed scheme to solve the conjunction of a stable and a non-stable condition is as follows: use the interval termination method until the first time is found where the stable condition is true, then use the rollback feature of time warp to restart simulation at the

last false point, and begin searching exhaustively for a time satisfying the termination condition from there. This is illustrated in Figure 9.

7.2 The Interval-Then-Exhaustive Termination Algorithm

Two UNITY programs are included in the Appendix D to solve the interval-then-exhaustive termination of a parallel simulation using the conjunction of a stable an a non-stable termination condition. The first program, *changes_to_tw_for_interval_then_exhaustive_termination,* is the superposed program that specifies what additions need to be made to the time warp algorithm given in Appendix A in order that processes running the time warp algorithm may contribute the data needed by the termination detector, and be terminated when the termination condition is true. The termination detector will be running the second program in Appendix E, *interval_then_exhaustive_termination* which will be unioned with the composite *time_warp_process.* This algorithm specifies what must be done a) to determine when the termination condition becomes true, b) to collect the output measures, and c) to make the UNITY program reach a fixed point.

8.0 DISJUNCTIVE TERMINATION CONDITIONS

In section 2.3, it was shown 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 two stable conditions, the disjunct of two non-stable conditions, and the disjunct of a stable and a non-stable condition.

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 two stable conditions. Once the two stable conditions become true they will remain true: thus the disjunct of two stable conditions is a stable condition. The disjunct of two non-stable conditions is similarly a non-stable condition, thus must be solved using the exhaustive method (see Table 2 on page 22).

8.1 The Union of a Stable and a Non-Stable Condition

The disjunct between a stable and a non-stable condition does not have a straightforward solution strategy. Figure 10 illustrates this possibility. Figure 10a represents a stable condition, 10b a non-stable condition, and 10c represents the disjunct of the two conditions. Whenever at least one of the conditions is true termination could occur.

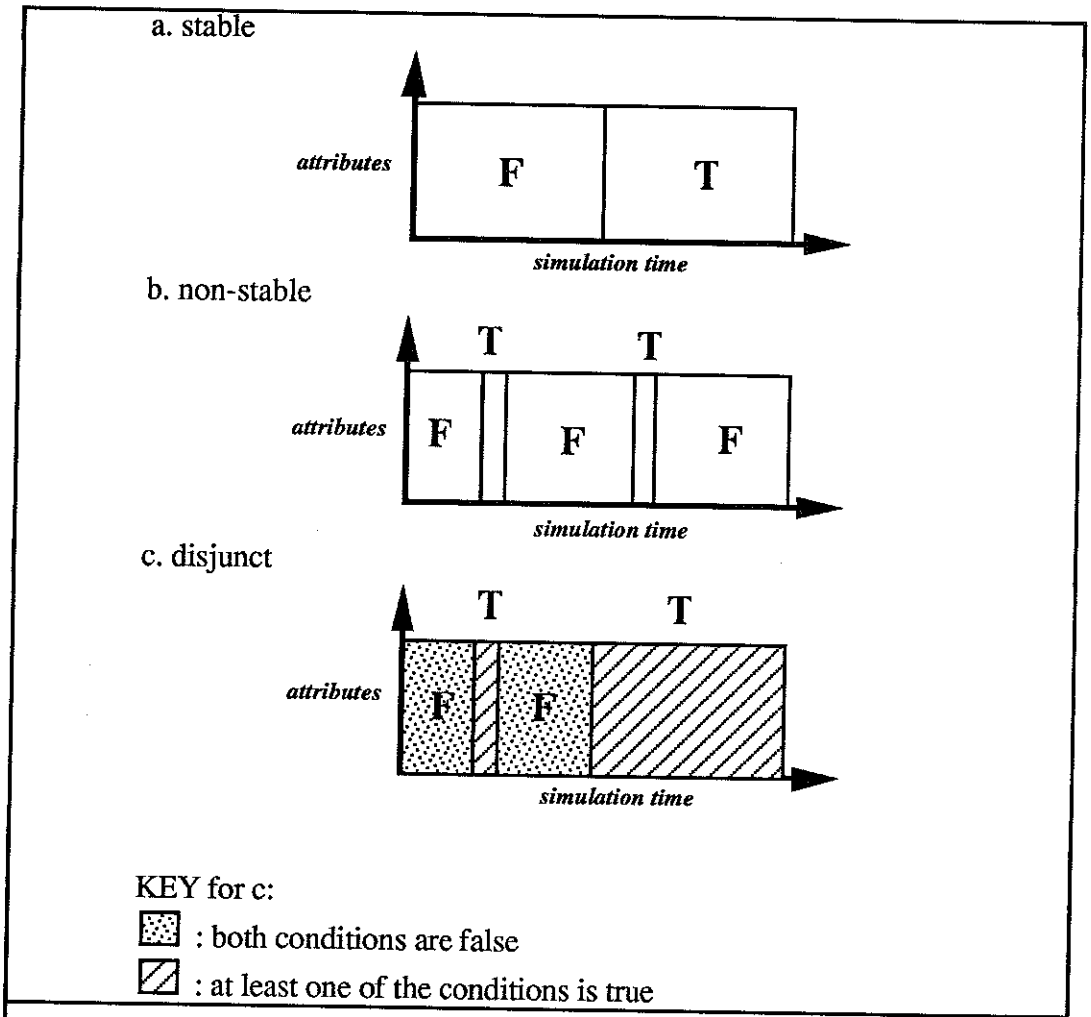

Figure 10: (c) is an illustration of the disjunct between a stable condition (a), and a non-stable condition (b). Whenever at least one condition is true termination can occur.

It would seem that if either the interval or the exhaustive termination methods could be used for termination detection, the interval method should be used (the argument for this is given in section 5.1). However it not guaranteed that the interval method will always be the most efficient solution. Take the example shown in Figure 11. In this case, if the

interval method is used (11a) termination will be detected long after it would be if the exhaustive method were used (11b).

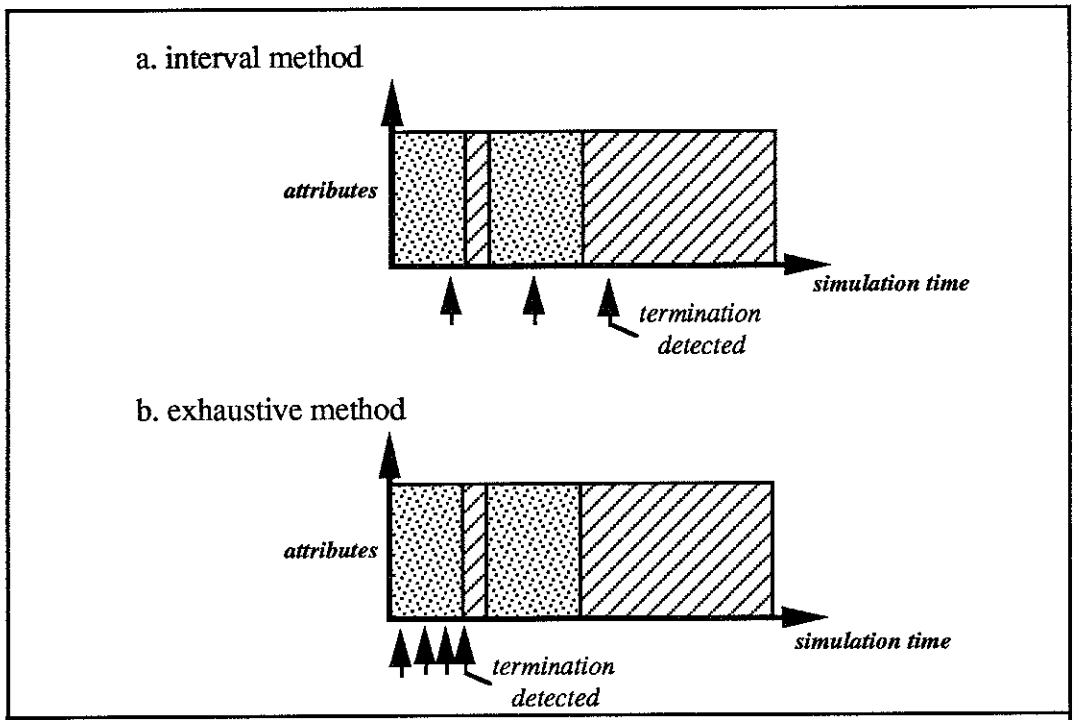

Figure 11: A possible scenario where the interval termination method (a) may detect termination later than the exhaustive termination method (b) would in the termination condition that is the disjunct of stable and non-stable conditions (see legend in Figure 10).

If a condition becomes true very soon after the simulation begins, the exhaustive method is likely to find where the termination condition can be satisfied before than the interval method could. However, if there is no a priori knowledge that this will be the case, the more likely choice for a simulation programmer to make would be to assume interval termination.

There is no clear recommendation on what method to use to detect the termination condition of a disjunct of stable and non-stable conditions.

	termination category (S stable, s not)	method for implementing

termination conditions	S	interval
	s	exhaustive
termination condition conjuncts	$S \wedge S$	interval
	$s \wedge s$	exhaustive
	$S \wedge s$	interval-then-exhaustive
termination condition disjuncts	$S \vee S$	interval
	$s \vee s$	exhaustive
	$S \vee s$	undetermined

Table 2: Method for implementing each termination category

9.0 CONCLUSION AND OPEN PROBLEMS

Determining when to terminate an asynchronous parallel simulation using a global termination condition is inherently complex. Because each time warp process 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. To determine *at what simulation time t* the global termination condition is satisfied when each process has a different t at a given wall clock time, and then collect the corresponding output measures, is not a trivial problem.

This paper presents a classification of termination conditions, and proposes a method to solve the termination detection problem when termination conditions that fall into each category are used. The value of this work would be to simulation programmers: both to implement any termination condition they want to choose for their simulation, as well as to guide their choice. If it fits a programmer's purpose equally well to use a stable termination condition or a non-stable one, he should be aware that implementing a non-stable termination condition is much more difficult and time consuming than implementing a stable termination condition.

Open problems remain. Most immediately, the UNITY algorithms given in this paper need to be proven for correctness. Beyond that, 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 processes cooperating in the time warp simulation is fixed).

ACKNOWLEDGEMENTS

The authors would like to thank Patrick Brown for his contribution regarding the time warp algorithm presented in Appendix A.

REFERENCES

Abrams, M. and D. Richardson (1990), "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation," VPI&SU tech rep 90-54, to appear in *Proceedings of the 1991 MultiConference on Simulation* (Anaheim, CA, Jan.).

Backus, J.W. (1959), "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference", in *International Conference on Information Processing,* June 1959, Unesco, Paris, pp 125-132.

Chandy, K.M. and J. Misra (1987), *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.

Jefferson, D. R. (1985), "Virtual Time," *ACM Transactions on Programming Languages & Systems 7*, pp 404-425.

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

APPENDIX A: TIME WARP ALGORITHM

Program *time_warp_process*
{this algorithm will run on all logical processes participating in the time warp simulation }
{ storage considerations have not been made; queues and arrays are infinitely large }
{ no code for calculating GVT and writing to GVTC is given }
{ time_warp_process largely contributed by Patrick Brown }

declare

```
type msg : record                          {form for input and output
             time : real                     messages. time indicates receive
             sign : boolean                  or send time respectively, sign
             value: integer                  indicates an antimessage, and
             end                             value could be interpreted as
                                             more than an int if need be}
type state: record                         {form to save in STATE array }
             time : real
             value: integer
             end
GVTC : sequence of reals                   {sequence of GVT values generated
                                             by some other process }
IC, OC : sequence of msg                   {sequence of input, output
                                             update msgs- channels}
IN : array [0..N'] of msg                  {to hold input msgs received by
                                             process }
OUT : array [0..N''] of msg                {to hold output msgs produced
                                             by process }
STATE : array [0..N'''] of state           {to hold states of the process }
curr_st,                                   {index to state array}
end_in,                                    {last input msg taken from IC }
ip,                                        {index used for IN ordering }
last_proc,                                 {index to most recent IN msg
                                             processed}
last_sent,                                 {index to most recent OUT msg
                                             sent}
next_msg : integer                         {index to most recent OUT msg
                                             produced.}
rollback_st,                               {boolean if state must be rolled
                                             back}
rollback_out : boolean                     {if OUT must be rolled back}
GVT,                                       {last GVT value removed from
                                             GVTC }
LVT,                                       {time of last IN msg processed }
rolltime_st,                               {time STATE must rollback to}
rolltime_out : real                        {time OUT must rollback to }
of(...) : function of type msg             {function of STATE[last_proc] to
                                             produce next OUT msg }
sf(...) : function of type state           {function of STATE[last_proc-1]
                                             and IN[last_proc] to produce
                                             next STATE's value and time }
```

```
initially
    curr_st, last_sent, next_msg = 0
    end_in, ip, last_proc = 1
    rollback_st, rollback_out = false
    rolltime_st, rolltime_out = infinity
    GVT, LVT = 0.0
    IN[0].time = -2, IN[1].time = -1          {to begin ordering so ip will
                                               behave correctly when no IN
                                               msg's have arrived yet. Done to
                                               simplify conditional in assign
                                               statement. }
assign
{take msgs from input channel}
    IN[end_in+1]    , end_in        , IC    :=
    head(IC)        , end_in + 1    , tail(IC)    if    IC <> null
{update GVT if update time arrives over channel}
    GVT             , GVTC    :=
    head(GVTC)      , tail(GVTC)              if    GVTC <> null

{input queue ordering}
{traverse down list if in correct order}
    ip  := ip - 1                             if    IN[ip].time > IN[ip-1].time
                                                ∧   IN[ip].time >= GVT
{wrap pointer back to top of list}
    ip  := end_in                             if IN[ip].time < GVT
{if msgs not in order and not processed yet, simply swap their positions in array IN}
 {if they are anti messages, annhialate both by replacing messages by null }
    ip      , IN[ip]      , IN[ip-1]  :=
    ip - 1  , IN[ip-1]    , IN[ip]            if    ip > last_proc + 1
                                                ∧   IN[ip].time <= IN[ip-1].time
                                                ∧   IN[ip].value <> IN[ip-1].value
    ip - 2  , null        , null              if    ip > last_proc + 1
                                                ∧   IN[ip].time = IN[ip-1].time
                                                ∧   IN[ip].sign <> IN[ip-1].sign
                                                ∧   IN[ip].value = IN[ip-1].value
{if msgs not in order and have been processed, must rollback}
    ip      , IN[ip]      , IN[ip-1]  , rollback_st  , rollback_out  , rolltime_st,
                                                rolltime_out, last_proc  :=
    ip -1   , IN[ip-1]    , IN[ip]    , true         , true          , IN[ip].time  ,
                                                IN[ip].time  , last_proc - 1

                                              if    IN[ip].time <= IN[ip-1].time
                                                ∧   ip <= last_proc + 1
{processing}
{process forward if there is an input msg to process and not rolling back}
```

```
STATE[curr_st], OUT[next_msg]      , curr_st       , next_msg  , last_proc   ,
                                                                    LVT          :=
sf(...)            , of(...)       , curr_st+1     , next_msg+1, last_proc+1,
                                                                    STATE[curr_st].time
                                     if      ¬ rollback_st
                                       ∧     ¬ rollback_out
                                       ∧     end_in > last_proc

[] {delete state if one to rollback}
   curr_st := curr_st - 1            if      rollback_st
                                       ∧     STATE[curr_st].time >=
                                                    rolltime_st

[] {rolled state back far enough}
   rollback_st, rolltime_st    :=
   false          , infinity         if      rollback_st
                                       ∧     STATE[curr_st].time <
                                                    rolltime_st

[] {output queue}
  {send msg if one to send and not rolling back}
   OC                          , last_sent     :=
   (OC;OUT[last_sent+1])       , last_sent + 1 if      ¬ rollback_out
                                       ∧     last_sent < next_msg - 1

[] {eliminate msg if it has not been sent and is to be rolled back}
   next_msg := next_msg - 1          if      rollback_out
                                       ∧     next_msg -1 > last_sent
                                       ∧     OUT[next_msg-1].time >=
                                                    rolltime_out

[] {rolled back far enough}
   rollback_out     , rolltime_out  :=
   false            , infinity      if      rollback_out
                                       ∧     next_msg - 1 > last_sent
                                       ∧     OUT[next_msg-1].time <
                                                    rolltime_out

[] {send antimessage for those sent after rollback time}
   OUT[next_msg-1].sign   :=  false
                                     if      rollback_out
                                       ∧     next_msg-1 <= last_sent

[]  OC                        , next_msg,     , last_sent     :=
   (OC;OUT[next_msg-1])       , next_msg - 1  , last_sent - 1
                                     if      rollback_out
                                       ∧     next_msg-1 <= last_sent
```

∧ OUT[next_msg-1].sign = false

end

APPENDIX B: INTERVAL TERMINATION ALGORITHM

Part 1 - Program to be superposed onto *time_warp_process*

Program *change_to_tw_for_interval_termination*
{the following details changes that need to be made to program time_warp_process *to incorporate* interval termination. *}*

add
declare
CTC : sequence of values (int) *{sequence of values needed by the term process to calculate the termination condition }*

tw_terminate : boolean *{this process will continue as long as this variable is false. Once it is made true by a message sent via TC no statement can occur, and the program has reached fixed point }*

always
st_lt_gvt = < max i : 0 <= i <= N ::
STATE[i].time < head(GVTC) >
{the state with the highest index whose time is still less than GVT. Used to send over CTC }

initially
tw_terminate = false

transform *{the following enumerates the needed changes to statements that exist in the original time_warp_process}*

for each statement s in the underlying program,
the additional condition if ¬ tw_terminate

transform the following statement s:

[] *{update GVT if update time arrives over channel}*
GVT , GVTC :=
head(GVTC), tail(GVTC) if GVTC <> null
∧ ¬ tw_terminate

to

{this sends the most recent state to the termination process.
st_lt_gvt defined in 'always' section }
s || CTC := STATE [st_lt_gvt]

end

Part 2 - Program to be unioned with the superposed *time_warp_process*

Program *interval_termination*
{algorithm to implement interval termination. Only tests the termination condition in intervals, the interval herein is when GVT is updated. }

declare

```
ATTR : array[1..N, 1..M] of integer          {array of attributes sent from each
                                               participating TW processes }
attr_ptr : array[1..N] of integer            {pointers to states saved from each
                                               participating TW processes }
cf(...) : function of type boolean           {this takes attribute values from each
                                               time warp process and determines
                                               if termination should occur }
GVT_term : real
gvt_num : integer                            {indicates the GVT currently being
                                               processed }
N : integer                                  {number of incoming CTC's from all
                                               participating TW processes }
opf(...) : function of type real             {takes all values and determines
                                               desired output measure }
output_measure : real                        {holds output measure calculated }
proc_tc : boolean                            {indicates if the term cond has been
                                               processed for the current GVT }
terminate : boolean                          {to indicate if TW processes can stop }
term_complete : boolean                      {so that opf(...) is only calculated once,
                                               and program will reach FP }
```

always

```
attributes = < i : 0 <= i <= N :: ATTR[ i, gvt_num] >
                                             {to use in functions cf and opf, the
                                               attribute value from each of
                                               N processes at the curr GVT }
have_attr_for_each = < min attr_ptr[i]  ::  attr_ptr[i] >= gvt_num  >
                                             {boolean to make sure each TW process
                                               has recorded an attribute for that
                                               time }
```

initially

```
GVT_term = 0.0
proc_tc = true
terminate = false
term_complete = false
< || i : 0 <= i <= N :: attr_ptr[i] = 0 >
```

assign

{ update gvt when msg comes in; indicate that the termination condition has not been calculated for this time }

```
    GVT_term      ,GVTC          ,gvt_num          , proc_gvt :=
    head(GVTC)    ,tail(GVTC)    ,gvt_num + 1,  false
                                                  if      GVTC <> null
                                                    ∧     proc_gvt
                                                    ∧     ¬ terminate

[] {take any incoming values from any TW process}
  <[] i : 0<=i<=N ::
      ATTR[ i, attr_ptr[i] ]    , attr_ptr[i]       , CTC[i]       :=
      head(CTC[i])              , attr_ptr[i] + 1   , tail(CTC[i])
                                                  if      CTC[i] <> null
                                                    ∧     ¬ terminate          >

[] {if all TW ps are accounted for for that GVT, calculate to see if termination can occur}
    terminate          , proc_gvt      :=
    cf( attributes )   , true                    if      have_attr_for_each
                                                   ∧     ¬ terminate

[] {if term can occur, write tw_terminate to all TW processs, and calculate output}
    tw_terminate       , output_measure , calc_complete  :=
    true               , opf( attributes ) , true
                                                  if      terminate
                                                   ∧     ¬ calc_complete     >
end
```

APPENDIX C: EXHAUSTIVE TERMINATION ALGORITHM

Part 1 - Program to be superposed onto *time_warp_process*

Program *change_to_tw_for_exhaustive_term_detection*
{the following details changes that need to be made to program time_warp_process *to incorporate* exhaustive_termination. *}*

add
declare
CTC : sequence of states (record of time and attribute value)
{sequence of values needed by the term process to calculate the termination condition }

send_ptr : integer
{indexing what states have been sent to the term process }

tw_terminate : boolean
{this process will continue as long as this var is false. Once it is made true by the termination process no statement can occur and the program has reached fixed point. }

initially
tw_terminate = false
send_ptr = 0

assign
{once GVT updated, send states before that time }
CTC , send_ptr := STATE[send_ptr], send_ptr + 1 if STATE[send_ptr].time < GVT

transform *{the following enumerates the needed changes to statements that exist in the origional time_warp_process}*

for each statement s in the underlying program,
the additional condition if ¬ tw_terminate

end

Part 2 - Program to be unioned with superposed *time_warp_process*

Program *exhaustive_termination*

{this procedure used to exhaustively test the termination condition after every attribute change by every TW process, used by non-stable termination categories }

declare

```
ATTR : array[1..N, 1..M] of state          {array of time & attributes sent
                                             from each participating TW
                                             process }
attr_ptr : array[1..N] of integer          {pointers to states saved from
                                             each participating TW process }
cf(...) : function of type boolean         {this takes all values and
                                             det's if term should occur }
N : integer                                {number of incoming CTC's from
                                             all participating TW processes }
opf(...) : function of type real           {takes all values and determines
                                             desired output measure }
output_measure : real                      {holds output measure calc'd }
prev_calc_time : real                      { last time calc'd cf(..) }
term_complete : boolean                    {so that once tw_terminate =
                                             true the progm will reach FP }
terminate : boolean                        {to indicate if TW ps's can stop}
```

always

```
attributes = < || i : 0 <= i <= N ::
                    < max j : ATTR[i,j].time <= next_calc_time ::
                                                ATTR[i,j].value >      >
                                               {for every i, use the value at the
                                                 time that is <= the time set to
                                                 calc,to use in functions cf and
                                                 opf, the attr value from each
                                                 of N processes at the curr time}
min_attr_time = < || i : 0 <= i <= N ::
                        < min ATTR[i, attr_ptr[i]-1].time > >
                                               {before can process the term
                                                 function, each TW ps must
                                                 have an ATTR from a equal or
                                                 higher time. This is to know
                                                 for sure that that ATTR will
                                                 hold for the span of time up to
                                                 the time to calc. }

next_calc_time = < || i,j: 0 <= i <= N, 0 <= j <= M ::
                        < min ATTR[i,j].time :: ATTR[i,j].time >
                                                          prev_calc_time > >
                                               {this finds the next smallest
                                                 time to calculate the term cond
                                                for }
```

initially

```
prev_calc_time = 0
term_complete = false
terminate = false
```

```
    < || i,j : 0 <= i <= N and 0 <= j <= M :: ATTR[i,j] = 0 >
    < || i : 0 <= i <= N :: attr_ptr[i], last_proc[i] = 0,0 >

assign
{take any incoming values from any TW process, merge into time list}
<[] i : 0<=i<=N ::
    ATTR[ i, attr_ptr[i] ]    , attr_ptr[i]          , CTC[i]      :=
    head(CTC[i])              , attr_ptr[i] + 1      , tail(CTC[i])
                                                       if    CTC[i] <> null
                                                         ∧   ¬ terminate          >

[] {if all TW processes are accounted for for that GVT, calc to see if term can occur}
    terminate          ,  output_measure   , prev_calc_time     :=
    cf( attributes )   , opf(attributes)   , next_calc_time
                                                       if    min_attr_time >
                                                                 next_calc_time
                                                         ∧   ¬ terminate

[] {if term can occur, write to the tw_terminate variable in all TW ps}
     tw_terminate      , term_complete   :=
    true               , true                  if    terminate
                                                 ∧   ¬ term_complete
end
```

APPENDIX D: INTERVAL-THEN-EXHAUSTIVE TERMINATION ALGORITHM

Part 1 - Program to be superposed onto *time_warp_process*

Program *change_to_tw_for_interval_then_exhaustive_term_detection*
{the following details changes that need to be made to program time_warp_process *to incorporate* interval_then_exhaustive_termination. *}*

add

declare

CTC : sequence of states (record of time and attribute value)
{sequence of values needed by the termination process to calc the termination condition }

exh_terminate : boolean
{the exhaustive termination method will continue as long as this variable is false. Once it is made true by the termination process no statement can occur and the program has reached fixed point. }

int_terminate : boolean
{interval term will continue as long as this variable is false. Once it is made true by the term ps exh term will occur. }

send_ptr : integer
{indexing what states have been sent to the term process }

send_states : boolean
{if can send updates to term process, made true when GVT updated }

always

st_lt_gvt = < max i : 0 <= i <= N''' ::
STATE[i].time < head(GVTC) >
{the state with the highest index which its time is still less than GVT. Used to send over CTC }

initially

int_terminate = false
exh_terminate = false
send_ptr = 0

assign

{once GVT updated, send states before that time, this is to do once interval term is completed, and are now doing exhaustive calcuations }
CTC , send_ptr :=

STATE[send_ptr], send_ptr + 1 if STATE[send_ptr].time < GVT
∧ int_terminate
∧ ¬ exh_terminate

transform *{the following enumerates the needed changes to statements that exist in the origional time_warp_process}*

for each statement s in the underlying program,
the additional condition if ¬ int_terminate
∧ ¬ exh_terminate

transform the following statement s:

{update GVT if update time arrives over channel}

GVT , GVTC :=

head(GVTC), tail(GVTC) if GVTC <> null
∧ ¬ int_terminate
∧ ¬ exh_terminate

to

{this sends the most recent state to the termination process. st_lt_gvt defined in 'always' section }

s || CTC := STATE [st_lt_gvt]

end

Part 2 - Program to be unioned with superposed *time_warp_process*

Program *interval_then_exhastive_termination*

{ The interval termination algorithm is done first, then once the term condition is found to be true, all TW processes are rolled back to a time when the term cond was false, and the exhaustive termination algorithm is run. }

declare

ATTR : array[1..N, 1..M] of integer *{array of attributes sent from each participating TW process }*

attr_ptr : array[1..N] of integer *{pointers to states saved from each participating TW process }*

cf(...) : function of type boolean *{this takes all values and determnes if term should occur}*

GVT_term : real

gvt_num : integer *{indicates the gvt currently being processed }*

int_complete : boolean *{a flag to show that the tw processes have been informed that the interval term part is complete }*

N : integer *{number of incoming CTC's from*

```
                                                    all participating TW processes }
    opf(...) : function of type real                {takes all values and determines
                                                      desired output measure }
    output_measure : real                           {holds output measure calc'd }
    prev_calc_time : real                           {last time calculated cf(..) }
    prev_gvt : real                                 {the last GVT calculated for, used as
                                                      lower bound for exhaustive
                                                      termination }
    proc_tc : boolean                               {indicates if the term cond has
                                                      been processed for the current
                                                      GVT }
    term_complete : boolean                         {so that opf(...) only calculated
                                                      once, and program reaches FP}
    term_exh : boolean                              {determines that the exhastive
                                                      calculaiton of the term cond has
                                                      been completed, thus the simulation
                                                      has reached fixed point }
    term_int : boolean                              {determines that the interval calcu-
                                                      lation of the term cond has been
                                                      completed }
    tw_msg : msg (record with fields time, sign, and value)
                                                    {this is used to cause rollback to
                                                      the point where the term cond
                                                      was false once a time is found
                                                      where it is true, sent to each
                                                      time warp process }

always
    exh_attributes = < || i : 0 <= i <= N ::
                        < max j : ATTR[i,j].time <= next_calc_time ::
                                                          ATTR[i,j].value > >
                                                    {for every i, use the value at the
                                                      time that is <= the time set to
                                                      calculate, to use in functions cf and
                                                      opf, the attribute value from
                                                      each of N processes at the current
                                                      time}
    have_attr_for_each = < min attr_ptr[i]   ::  attr_ptr[i] >= gvt_num  >
                                                    {boolean to make sure each TW
                                                      process has recorded an attribute for
                                                      that time }
    int_attributes = < i : 0 <= i <= N :: ATTR[ i, gvt_num] >
                                                    {to use in functions cf and opf,
                                                      the attribute value from each of
                                                      N processes at the current GVT }
    max_st_lt_time = < max i : 0 <= i <= N''' ::
                        STATE[i].time <= tw_msg.time >
                                                    {pointer to the state with the
                                                      highest index number that is
                                                      less than the rollback time }
    min_attr_time = < || i : 0 <= i <= N ::
                          < min ATTR[i, attr_ptr[i]-1].time > >
                                                    {before can process the termination
                                                      function,  each TW process must have
```

an ATTR from a equal or higher time. This is to know for sure that that ATTR will hold for the span of time up to the time to calculate. }

next_calc_time = < || i,j: 0 <= i <= N, 0 <= j <= M ::
< min ATTR[i,j].time :: ATTR[i,j].time > prev_calc_time > >
{this finds the next smallest time to calculate the term cond for }

initially

GVT_term = 0.0
int_complete = false
prev_calc_time = 0
proc_tc = true
term_complete = false
term_exh = false
term_int = false
tw_msg.sign = true
tw_msg.value = null *{effect of receipt of this msg will be only rollback of time}*
< || i,j : 0 <= i <= N and 0 <= j <= M :: ATTR[i,j] = 0 >
< || i : 0 <= i <= N :: attr_ptr[i] = 0 >

assign

{ update gvt when msg comes in, indicate that the term cond has not been calculated for this time }

GVT_term ,GVTC ,gvt_num , proc_gvt , prev_gvt :=
head(GVTC) ,tail(GVTC) ,gvt_num + 1 , false , GVT_term
if GVTC <> null
∧ proc_gvt
∧ ¬ term_int
∧ ¬ term_exh

[] *{take any incoming values from any TW process}*

< [] i : 0<=i<=N ::
ATTR[i, attr_ptr[i]] , attr_ptr[i] , CTC[i] :=
head(CTC[i]) , attr_ptr[i] + 1 , tail(CTC[i])
if CTC[i] <> null
∧ ¬ term_int
∧ ¬ term_exh >

[] *{if all TW ps are accounted for for that GVT, calculate to see if termination can occur, store the time in message field in case cf is true & msg will be sent next }*

term_int , proc_gvt , tw_msg.time :=
cf(int_attributes) , true , prev_gvt
if have_attr_for_each
∧ ¬ term_int
∧ ¬ term_exh

```
[] {if termination can occur, must cause rollback on all participating TW processes to the
   last time cf() was calculated and was false to begin exhaustive search.
   All of these written to variables except int_complete were defined in the
   TW algorithm and must be altered in each participating TW process.  }
   int_complete   , GVT        , int_terminate   , IN[end_in+1]    , end_in     ,
                                                      send_ptr                  :=
   true           , prev_gvt   , true            , tw_msg         , end_in + 1,
                                                      max_st_lt_time
                                                 if   ¬ int_complete
                                                  ∧   term_int
                                                  ∧   ¬ term_exh              >

[] {take any incoming values from any TW process, merge into time list}
   <[] i : 0<=i<=N ::
      ATTR[ i, attr_ptr[i] ]   , attr_ptr[i]       , CTC[i]      :=
      head(CTC[i])             , attr_ptr[i] + 1   , tail(CTC[i])
                                                 if     CTC[i] <> null
                                                  ∧     term_int
                                                  ∧   ¬ term_exh    >

[] {if all TW processes are accounted for for that GVT, calc to see if term can occur.
   Output_measure is calculated at this time to ensure the same attribute
   values are used for cf() and opf(). }
   term_exh              , output_measure     , prev_calc_time     :=
   cf( exh_attributes )  , opf(exh_attributes)  , next_calc_time
                                                 if     min_attr_time >
                                                            next_calc_time
                                                  ∧     term_int
                                                  ∧   ¬ term_exh

[] {if termination can occur, write to the variable in all TW processes}
   <[] i : 0<=i<=N ::
     exh_terminate, term_complete   :=
     true         , true                       if     term_exh
                                                  ∧     term_int
                                                  ∧   ¬ term_complete    >
end
```