Order Number 9521287

An implementation of GPSS as a PDES with roll-back on a parallel computer

Kim, Jungmi Yoon, Ph.D.

City University of New York, 1995

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

An Implementation of GPSS as a PDES with Roll-Back on a Parallel Computer

by

JUNGMI YOON KIM

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1995

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Jan 23, 1995 — Date

Chair of Examining Committee

Jan 24, 1995 — Date

Executive Officer

Professor Nabil Adams

Professor Seyed-Ali Ghozati

Professor Stanley Habib

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

AN IMPLEMENTATION OF GPSS AS A PDES WITH ROLLBACK ON A PARALLEL COMPUTER

by

JUNGMI YOON KIM

Advisor : Professor Theodore Brown

Parallel Discrete Event Simulation (PDES) is the execution of a single discrete event simulation on a parallel computer. Simulation has always been an important tool in the modeling tool kit because it is one of the easiest modeling techniques to apply and the fact that simulation models can capture the actual characteristics of the system being modeled readily.

Recent studies on simulation methodology have been carried out in various directions. In particular, parallel simulation, the topic dealt with in this thesis, has had growing interest recently. Parallel computer systems can speed up a program's execution by making use of several computers simultaneously. The implementation of PDES causes synchronization conflicts(out-of-order-execution), which are generally pointed out as the most important problem to be solved.

This thesis is concerned with presenting ways of

implementing PDES, based on GPSS statements. Specifically, this thesis will show how to implement GPSS as a PDES using C-LINDA, a parallel programming language, while solving synchronization conflicts that occurs in the implementation process. In the process, it suggests some practical ways of how to implement PDES using GPSS.

Thus, the contribution of this thesis is to allow a wider community of users to gain access to parallel simulation.

ACKNOWLEDGMENTS

I would like to express my heartfelt thanks to my thesis committee members : Professors Nabil Adams, Theodore Brown, Seyed-Ali Ghozati and Stanley Habib for their interest and participation in my doctoral defense.

I am grateful to Professor Brown, my thesis supervisor. After taking his first class, I received new interest in simulation. He introduced me to a broad and deep knowledge of parallel simulation. Professor Brown provided me with all the theoretical background that I needed to implement GPSS as a PDES. He was always ready and willing to discuss my dissertation and tried to understand my meaning and intention.

I am also thankful to the many people who helped me with my dissertation, Myra Mniewski, Bill Vanyo and Prashant Joshi. Myra spent lots of time with me editing my dissertation.

I especially want to thank my husband for the tremendous moral and spiritual support he offered me. This would not have been possible without my parents-in-law who took care of my daughters during my study. I shall never forget them.

I would also like to thank my parents, brothers, sisters and relatives for their emotional and financial support, and for their constant prayers for my successful completion of my doctoral degree.

Finally, I regret having to be separated from my daughters to complete this work, but I am grateful to them for their presence and for managing so well without me.

- C O N T E N T S -

FIGURES AND TABLES

Chapter 1. Introduction

1. The purpose of study

In organizational settings, we are often confronted with complex decision problems. To help solve such problems, a lot of techniques such as, simulation, statistics, linear programming, stochastic processes, network analysis, decision theory, queueing theory, inventory control, and so on, have been developed. Among these techniques, simulation has particularly attracted a considerable amount of interest because simulation models can most easily be realistic, in the sense of capturing the actual characteristics of the system being modeled.

Recent studies on simulation methodology have been carried out in various directions. In particular, parallel simulation, the topic dealt with in this thesis, has had growing interest recently. Parallel computer systems can make speed-up possible in the program execution by making use of several computers simultaneously. However, the implementation of Parallel Discrete Event Simulation(PDES) causes synchronization conflicts, which are generally considered the most important problem to be solved to take full use of PDES. To attack the problem, two general approaches have been developed: one is conservative which avoids the possibility of executing events out-of-timestamp-order, the other is optimistic which uses a detection and recovery approach. The optimistic approach

determines when an out-of-order-execution has occurred, and invoke a procedure to recover. This is called rollback.

This thesis, focusing on optimistic PDES, is primarily concerned with finding a way to implement PDES, without needing thorough understanding about parallel processing. Specially, this thesis suggests a way to implement PDES based on GPSS programs.

The basic idea of this thesis for the implementation of optimistic PDES in a GPSS way is as follows :

A GPSS like simulation program using C-LINDA is decomposed into several processes referred to as workers, each of which can be run on its own processor. Another process, the control process, referred to as master is required to generate the tasks, check for synchronization problems, and gather and report the results. The master process maintains a linked list ordered by time of the state of the system each time a statement that includes a global variable is executed. In order to make roll-back possible, each worker process must maintain a linked list that includes a history of its own transaction movements and a future events list. A tuple, like the local time of the interrupt worker process, is stored in the tuple space to use when each worker process communicates with the master process for checking the common passive resources and the termination counter. The master process starts each worker process, all of which then execute simultaneously.

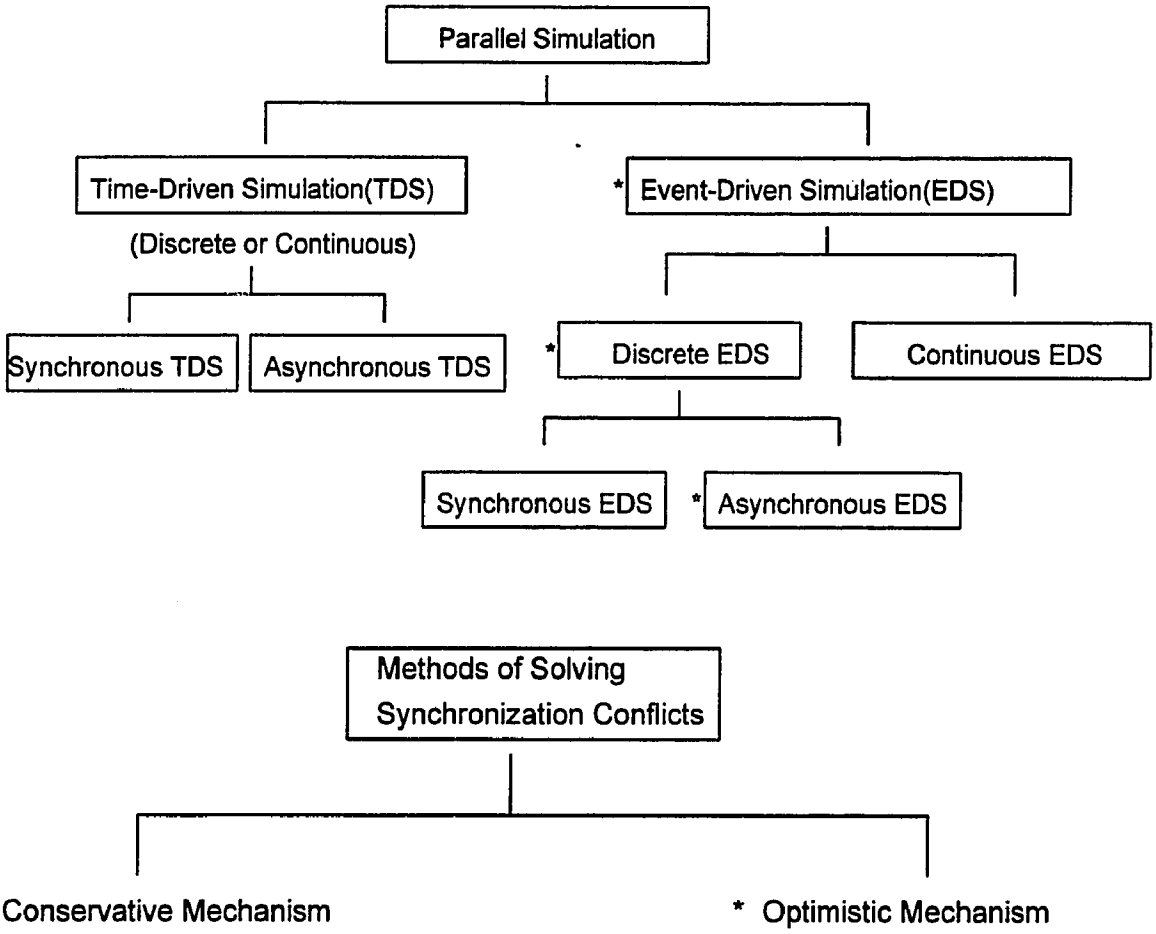
Whenever a worker process wants to use a common passive resource, it checks if a synchronization problem has occurred or not using the tuples in the tuple space. Interprocess communications between processes is by tuples in the tuple space. Whenever a process requests some information from the tuple space, the process gets the tuples from the tuple space using the IN or RD function of C-LINDA. Also, if a process wants to change some information in the tuple space after checking a synchronization problem, then the process updates the tuples in the tuple space using the OUT function of C-LINDA. If the synchronization problem (out-of-order-execution) has occurred, then all prematurely executed events have to be rolled back to the time just before the conflict, using the ascending linear list in every worker process. Forward events in the linear list are destroyed. And local time is reset. If a synchronization problem has not occurred, each worker process continues to run without interruption.

Accordingly, it is hoped that this implementation not only solves the synchronization conflicts occurring in the implementation of PDES, but also eventually contributes to a wider audience's access to parallel simulation.

2. The Organization of the Thesis

The main stream of parallel simulation studies

attained up to present is as follows :



As shown above, the studies on parallel simulation have been carried out in various directions. This thesis will deal with optimistic asynchronous event-driven parallel simulation using C-LINDA.

The rest of this thesis is organized as follows. Chapter 2 describes the general concept of PDES and the methods to solve the synchronization conflict problem. Also, the Time Warp mechanism, the most well known optimistic protocol, is introduced. This chapter includes a

literature review concentrating on the recent results of parallel simulation.

Chapter 3 presents an implementation of some GPSS statements for a parallel simulation using C-LINDA. Each statement of GPSS is checked to solve the synchronization conflicts and C-LINDA is introduced. This chapter suggests some algorithms for parallel simulation.

Chapter 4 shows an experiment involving the implementation of GPSS as a PDES using C-LINDA. This chapter shows how to change the GPSS program to the parallel program using C-LINDA and how to use the suggested algorithms. It includes the analysis of results.

Chapter 5 concludes with a discussion of this thesis and possible future work.

Chapter 2. The Review of the Parallel Discrete Event Simulation (PDES)

1. Definition

(1) Discrete Event Simulation

Discrete event simulation is a simulation in which the system being simulated only changes state at discrete points in simulated time. That is, the simulation time is incremented by event time (timestamped time), where an event represents a change in state. The simulation clock is advanced after the simulation of an event to the time of the next event[6].

(2) Parallel Discrete Event Simulation

Parallel discrete event simulation refers to use of more than one computer to simultaneously execute a single discrete event simulation. In PDES[7], a physical system has a finite number of physical processes that we call PPs. Each PP can be operated independently, except when it interacts with other PPs in the system. The interaction is by messages. Message lines connect one PP to another.

A logical system is a group of logical processes, denoted by LPs, that is topologically isomorphic to the physical system it simulates, with each PP being replaced and simulated by one LP.

In an asynchronous PDES, each LP has its own local clock that represents the simulated time. The simulated time for each LP is advanced to the minimum next event time for that process. Consequently, different LPs may be at different simulated times. Each LP is executed simultaneously, even if they are at different simulated times. The results of the parallel simulation are equivalent to a sequential simulation if the super imposed execution of events would be identical to the execution on a single computer.

2. Problems and Solutions with PDES

A principal issue in PDES is synchronization. Each LP must execute events in non-decreasing timestamp order. In PDES, events in each LP may be not processed in non-decreasing timestamp order. It is the synchronization mechanism's responsibility to ensure that each LP processes events in timestamp order. There are two general approaches to attack the synchronization problem : one is conservative which assures the monotone non-decreasing timestamp order, and the other is optimistic, which permits processors to execute events in non-monotonic timestamp order. If events are executed out of timestamp order, a rollback is needed. For example, if an event that needs to be executed at time t1 arrives to the processor after its clock is at t2, t2 > t1, rollback to t1 is needed. The result of an optimistic

simulation after roll-back is the same as a conservative simulation. That is, the optimistic mechanism detects a synchronization conflict(out-of-order-execution) and invokes a roll-back procedure to recover, while the conservative mechanism avoids the possibility of executing events out of timestamp order.

In this chapter, the difference between conservative mechanism and optimistic mechanism is considered by using the example in Fig.1 taken from [30].

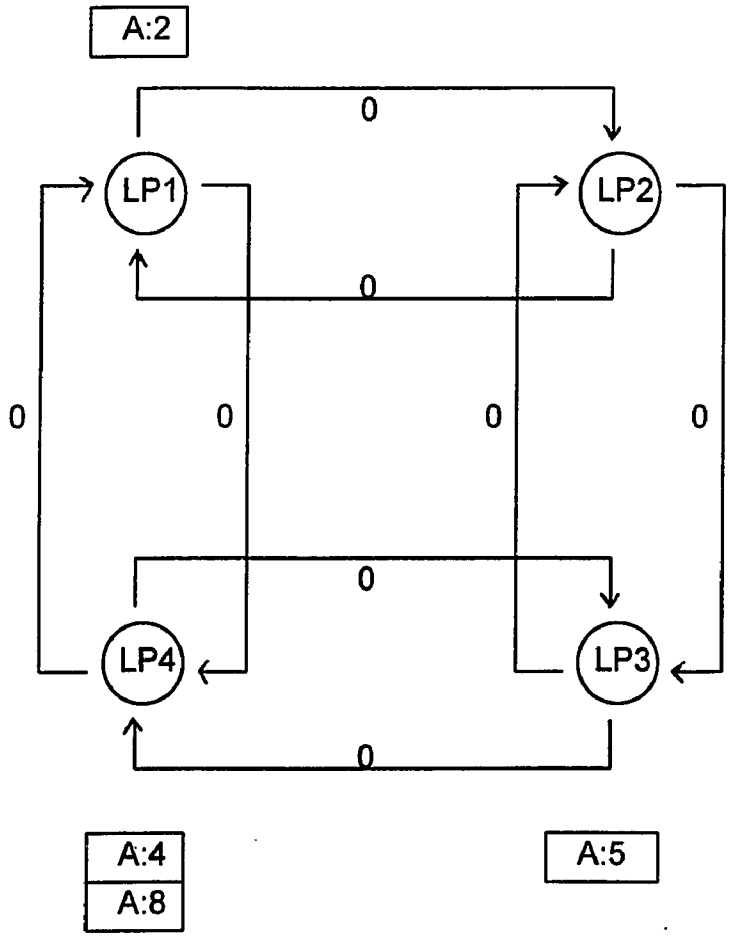


Fig.1. Initial Condition

As in fig.1, consider the network of four LPs. Each LP may route an event to one of two other LPs. Each LP maintains a list of events; in the Fig.1, A:4 denotes a job arrival event scheduled for time 4. Values on communication arcs (link times) denote the time stamp of the last message sent over that arc. Suppose that each LP is simulated on its own processor. And suppose that the service time of any job is at least 0.1. At the beginning of the simulation, a LP knows its initial job arrival time (presumably placed there as part of initialization), and arc times are initialized to zero.

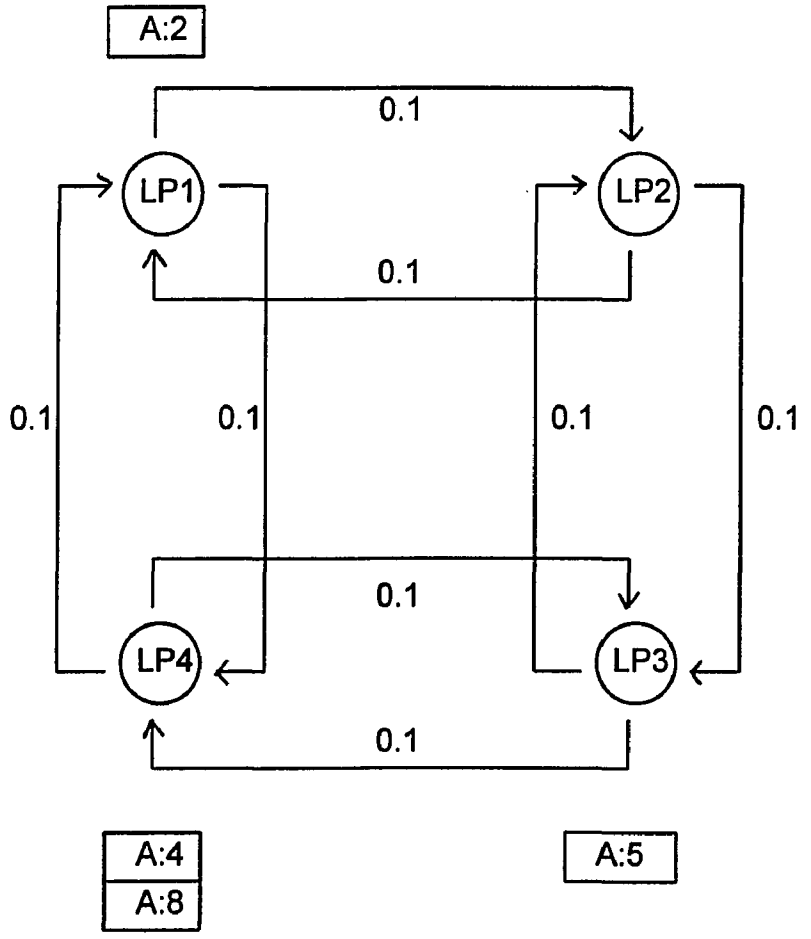
(1) Conservative Mechanism

The conservative mechanism requires that messages from any process to any other process be transmitted in chronological order according to their timestamps. That is, no LP can simulate its first event until it is certain that it will not receive a routed job with a timestamp less than its first arrival time.

PROBLEM : No LP can simulate its first event because the arrival times are all strictly greater than the initialized link times.

SOLUTION : One scheme to resolve this uses null-messages. Every LP reasons "even if I were to receive a job at time 0, that job would require at least 0.1 service time, therefore I can promise not to send a job until at

least time 0.1"; this reasoning permits the LP to send a null-message with timestamp 0.1 to both LPs to which it routes jobs. Since every LP does this, every link time eventually increases to 0.1 as in Fig.2. Under the CMB (Chandy-Misra-Bryant) rules the LP may receive and process the message associated with the least link time. Eventually a LP receives two null-messages with the same timestamp, and these may be processed. As a result, each LP sends two new null-messages, now with timestamp 0.2. This sort of gradual escalating of null-message timestamps continues



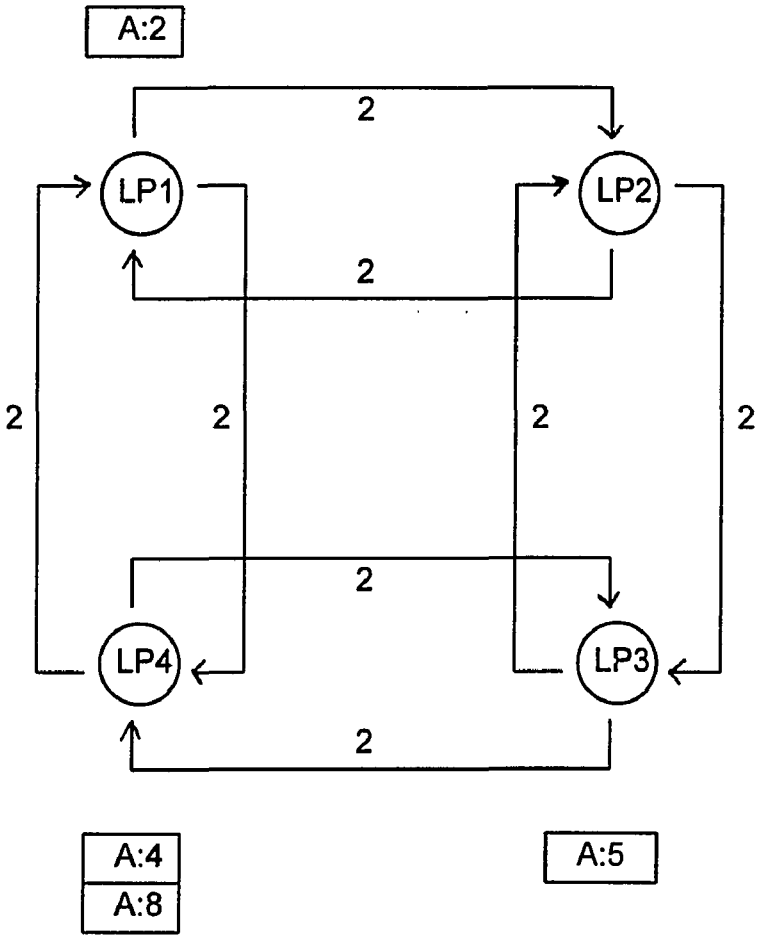


Fig.2. Using null-message

until the link times increase to the point of the LP1 arrival at time 2. At this point actual simulation activity begins. Observe that twenty rounds of null-message increments were needed just to reach this point. Suppose the LP1 arrival goes into service, is non-preemptable, and will depart at time 3. Knowing this, LP1 can send null-messages with timestamp 3 ("looking ahead" to the job's completion) to LP2 and LP4, leading to the situation illustrated in Fig.3. Continued incremental advances in null-message timestamps are needed to raise link times to a

high enough level so that the LP1 departure at time 3 can be simulated. The problem with this scheme is clearly the high volume of null-messages.

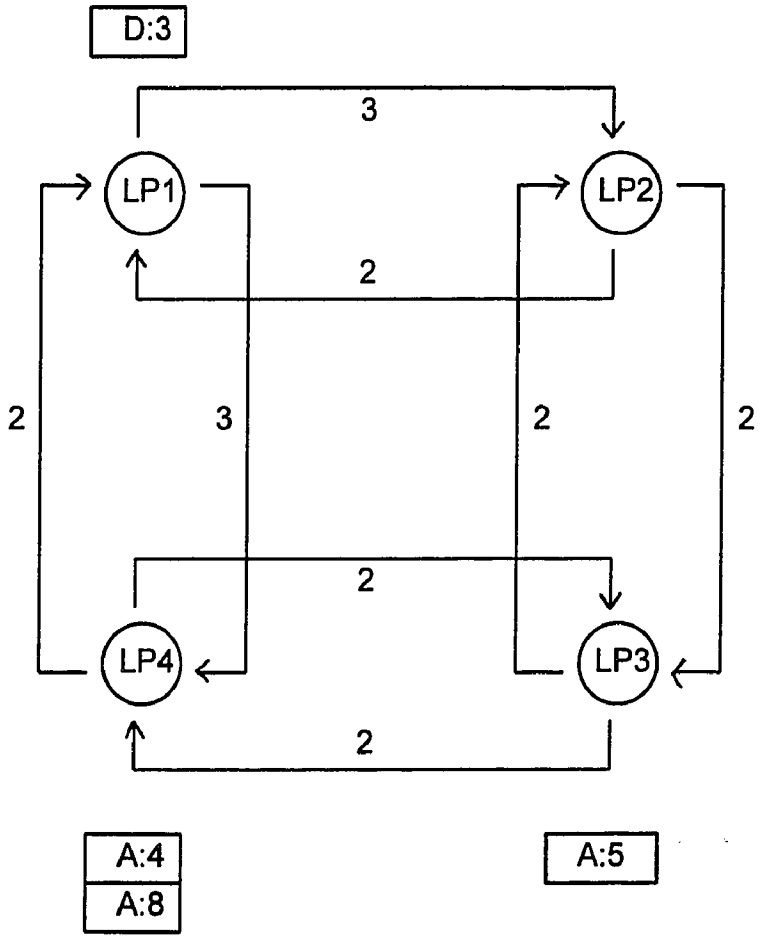


Fig.3. After first event simulation

(2) Optimistic Mechanism

In the optimistic mechanism, a process' clock may run ahead of the clocks of its incoming links and, if errors are made in the chronology, time must be "rolled

back" to correct them. Every LP checkpoints its state, then optimistically executes the first event.

PROBLEM : Suppose that the LP1 arrival at time 2 departs at time 3 and the job is routed to LP4. If LP4 has already simulated an arrival at time 4 when LP1 sent the job to LP4, error is made in chronology as in Fig.4.

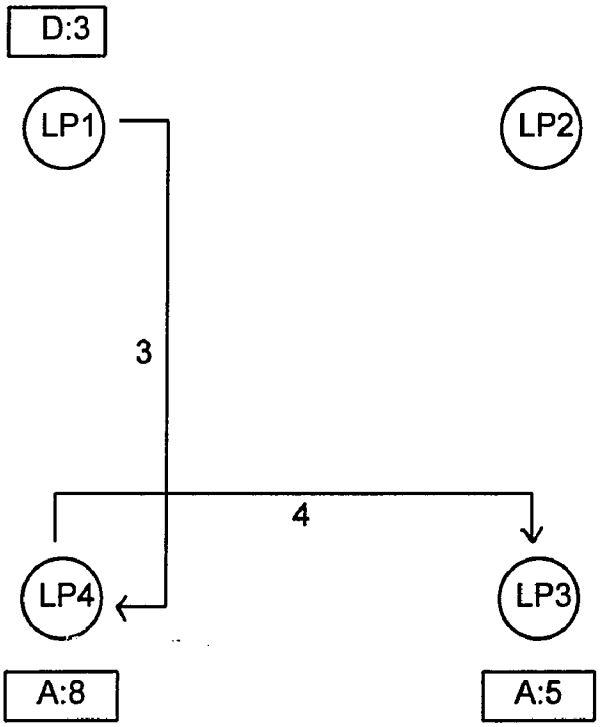


Fig.4. After first event simulation

SOLUTION : Since LP4 has already simulated an arrival at time 4, it must now be undone, along with all messages that may have been sent after to time 3. By the Time Warp roll-back mechanism, it send anti-messages after messages it erroneously sent. It recovers its initial state and simulates the new arrival. If LP3 has already simulated

an arrival at time 5, it too must roll-back.

3. Optimistic PDES

(1) Time Warp

In this chapter, an optimistic approach to asynchronous distributed simulation called Time Warp, the most well-known optimistic protocol, is explained. Time Warp was proposed by Jefferson and Sowizral[21].

In the Time Warp mechanism, a simulation consists of a set of LPs. These LPs interact with each other through event messages. Each LP has a local virtual time(LVT) which acts as the simulation time for that LP and a single input queue of event messages that acts as the LP's local event queue. LPs execute asynchronously because incoming event messages may not arrive at an LP in increasing timestamp order. Processes will often receive event messages with a timestamp that is less than the process' LVT. These are referred to as stragglers. An LP that receives a straggler event is rolled back to a LVT before the timestamp of the straggler event. In order to roll-back, a process must save copies of all events and be able to undo their effects.

1) The Organization and the Control Mechanism[20,21]

Under Time Warp, an event message has six components :

. The "send time" represents the LVT of the sender at the moment of sending. It determines the message's position in the sender's output queue.

. The "receive time (timestamp)" is the simulation time of the event. It determines the message's position in the receiver's input queue.

. The "sender" is the name of the LP sending the message.

. The "receiver" is the name of the receiving LP.

. The "sign" is a bit indicating whether the message is positive (indicated by +) or negative (indicated by -).

. The "text" is an event information indicating the kind of event, along with any parameter values needed for the simulation of the event.

An LP has five components:

. The "LVT register" which acts as the LP's local simulation clock holds the LVT for this LP.

. The "current state" holds the variables representing the state of the LP at LVT.

. The "input message queue" holds input messages ordered by virtual receive time.

. The "output message queue" holds copies of messages ordered by the sending virtual time.

. The "state queue" holds snapshots of some of the LP's past states ordered by LVT in order to make roll-back possible.

The structure of a event message and an LP is depicted as in Fig.5 and Fig.6.

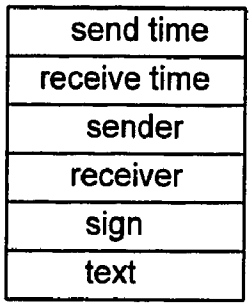


Fig. 5. The Structure of an Event Message

An anti-message is a copy of a previously sent message that differs from the original in only a sign bit field. Whenever an LP sends a message, a copy of the message is inserted in the receiver's input queue by the its virtual receive time and a negative copy is saved in the sender's output queue by its virtual send time. And whenever a message arrives to a LP with receive time greater than LVT, it is inserted in the input queue at the position determined by its virtual receive time. The LP continues processing messages from its input queue. If a message arrives to an LP with receive time less than LVT ("straggler"), the LP must roll-back to the receive time of the straggler.

The mechanism used to roll-back an LP is the heart of the Time Warp mechanism. If an LP receives a straggler, the LP sets the current state to the last state saved before the receive time of the straggler, and sets

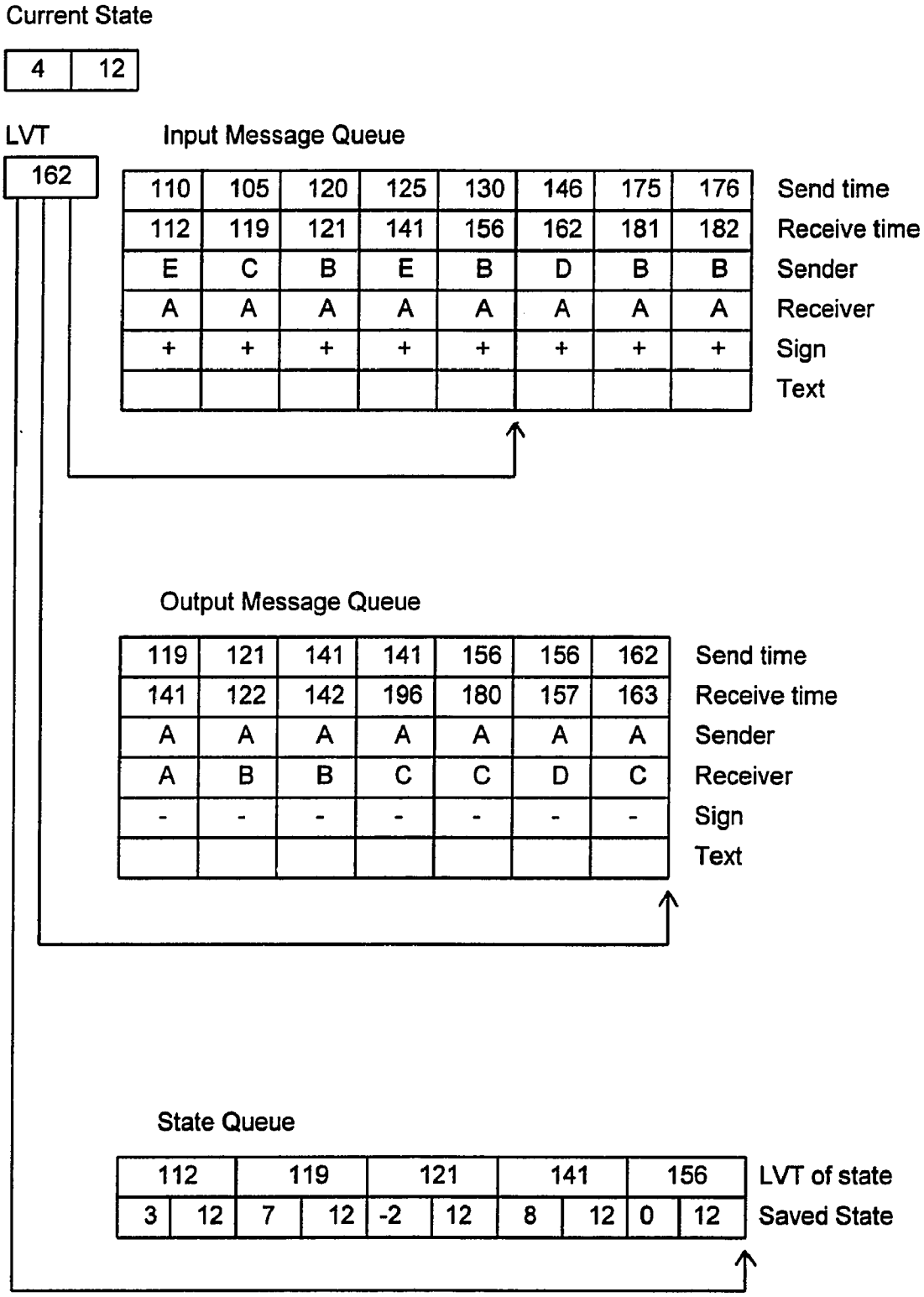


Fig 6. The Structure of an LP

its LVT to the time of that state. All later states in its state queue are removed. The LP sends all anti-messages from its output queue that have send times greater than its LVT. Whenever an anti-message (negative message) and a positive message are both in the same queue, the two annihilate each other, leaving no trace behind. Then the LP processes the straggler and continues to forward messages in its input queue.

In aggressive cancellation, whenever a straggler arrives at an LP, the LP immediately sends anti-messages from its output queue and cancels all messages that have send times later than its new LVT. In lazy cancellation, anti-messages are not sent immediately after roll-back. The LP resumes executing forward and waits until the LP checks if a produced message is different from messages in its output queue. Only messages that are different from previously sent messages are transmitted.

In Fig.7, the arrival of a straggler and the consequent roll-back for Fig.6 are shown.

(2) Previous Studies[13,14]

In this section, the development of parallel simulation is traced by reviewing the literature concentrating on recent works. The works are classified by protocols, hardware support, load balancing, memory management, time parallelism, and analytical performance

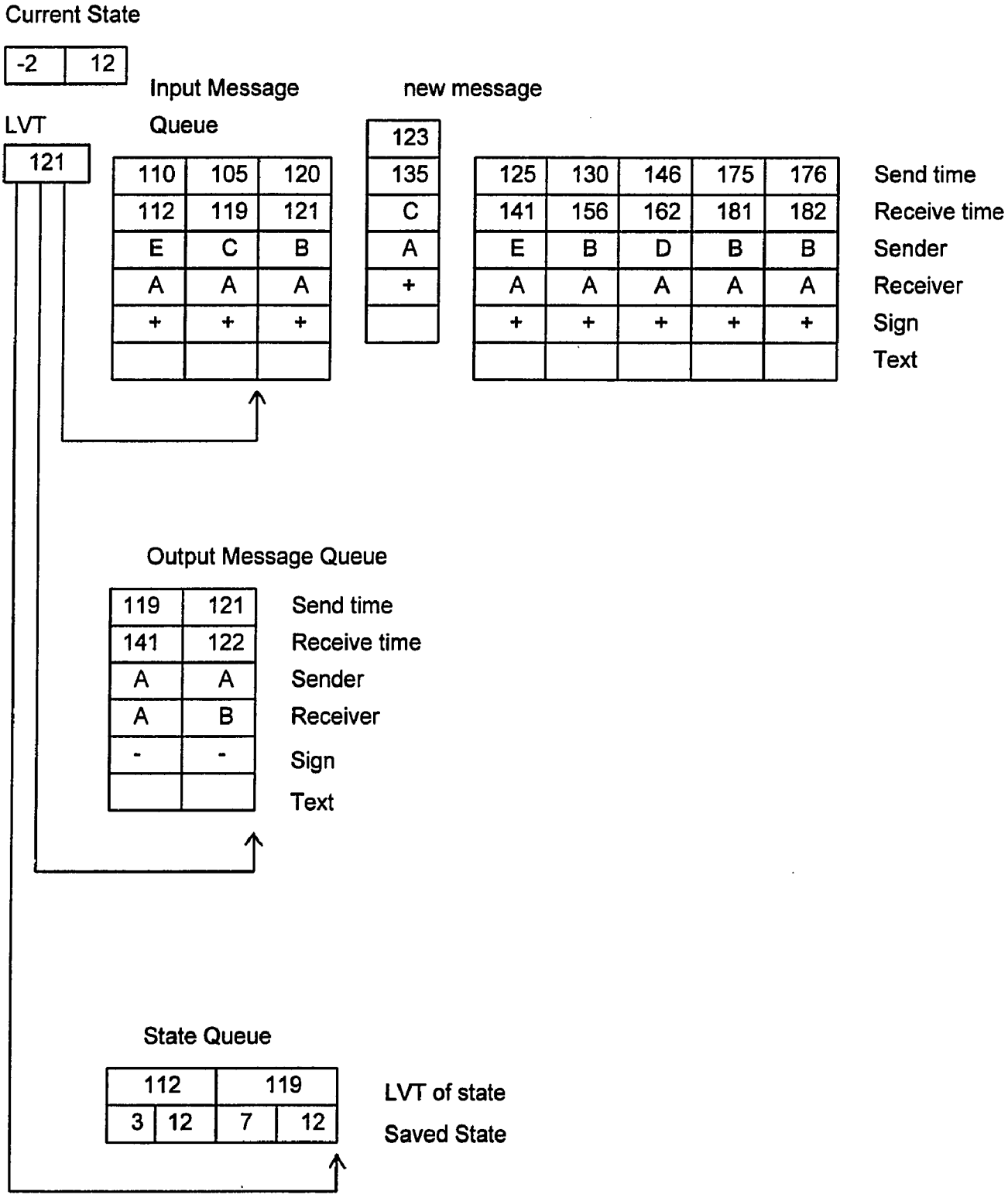


Fig.7. After Roll-back

analysis.

1) Protocols

. Jefferson(1985) and Sowizral(1982) used the Virtual Time paradigm, implemented by the Time Warp mechanism with aggressive cancellation.

. Gafni(1988) also used the Time Warp mechanism with lazy cancellation.

. West(1988) used the lazy reevaluation optimization (jump forward) which is similar to lazy cancellation, but deals with state vectors rather than messages. This requires a comparison of state vectors to determine if the state has changed.

. Sokol, Briscoe and Wieland(1988) used Time windows to prevent incorrect computations from propagating too far ahead into the simulated time future.

. Madisetti, Walrand and Messerschmitt(1988) proposed a mechanism in Wolf whereby a straggler message causes a process to send special control messages to quickly stop the spread of erroneous computations. Processes that may be "infected" by the erroneous computation are notified when an error(i.e., a straggler message) is detected.

. Fujimoto(1989) proposed a direct cancellation mechanism which uses shared memory to optimize the cancellation of incorrect computations. Whenever an event E1 schedules another event E2, a pointer is left from E1 to E2. This pointer is used if it is later decided that E2

should be canceled(using either lazy or aggressive cancellation).

. Chandy and Sherman(1989) suggested that the simulation can be viewed as a two dimensional space-time graph where one dimension enumerates the state variables used in the simulation, and the second dimension is simulated time.

. Turner and Xu(1992), Ball and Hoyt(1990), and Lubachevsky(1989) proposed another line of research that is to constrain Time Warp's optimism.

2) Hardware Support

. Fujimoto(1992) proposed a component called the roll-back chip that provides hardware support for state saving and roll-back in Time Warp.

. Buzzell, Robb and Fujimoto(1990) developed a prototype implementation of the roll-back chip in the commercial sector.

. Ghosh and Fujimoto(1991) extended the roll-back chip work to support a timestamp addressed memory system called space-time memory which is the principal component of a machine architecture called the Virtual Time Machine that uses roll-back as the principal primitive for synchronization.

. Reynolds(1991) and Pancerella(1992) proposed a hardware mechanism to rapidly collect, operate on, and disseminate synchronization information throughout a

parallel simulation system.

. Gopalakrishnan and Fujimoto(1991) verified the hardware design of the roll-back chip using formal techniques.

3) Load Balancing

. Nicol and Reynolds(1985) found early work on static and dynamic load balancing. Static load balancing algorithms distribute a fixed set of processes over the processors in the system. Dynamic algorithms allow processes to migrate during the execution of the parallel simulation.

. Briner(1990) examined static partitioning algorithms for digital logic simulation, based on Time Warp.

. Reiher and Jefferson(1990) proposed a new metric called effective processor utilization which is defined as the fraction of the time during which a processor is executing computations that are eventually committed.

. Glazer(1992) allocated virtual time-slices to processes, based on their observed rate of advancing the local simulation clock.

4) Memory Management

. Jefferson(1985) provided a fossil collection mechanism to reclaim "old" history information that is no

longer needed.

. Jefferson(1985) first proposed a mechanism called message sendback. In message sendback, the Time Warp executive may return a message to its original sender without over processing it, and reclaim the memory used by the message.

. Gafni(1988) proposed a protocol that utilizes message sendback as well as other mechanisms to reclaim storage used by state vectors and messages stored in the output queue when a process finds that its local memory is exhausted.

. Lin and Lazowska(1989) proposed a scheme that avoids acknowledgments by having each process communicate with the other processes to which it communicates when it begins a GVT computation in order to identify any transit messages.

. Jefferson(1990) proposed an alternative approach called cancelback. While Gafni's algorithm will only discard state in the process that ran out of memory, cancelback allows state in any process to be reclaimed. Messages containing high send-timestamps are sent back to reclaim storage allocated to messages. This tends to roll-back processes that are ahead of others in the simulation.

. Lin(1992) proposed the artificial roll-back algorithm. When storage is exhausted and fossil collection fails to reclaim additional memory, processes are rolled back to recover memory.

. Nicol(1992) developed a barrier algorithm for optimistic computations that can effectively serve to compute GVT.

5) Time Parallelism

. Chandy and Sherman(1989) observed that simulations are fixed-point computations, and as such can be executed as asynchronous-update computations.

. Greenberg(1991) established practical exploitation of time parallelism.

. Heiedelberger and Stone(1990), Ammar and Deng(1991) and Lin and Lazowska(1991) suggested a more direct approach to time parallelism that is to partition the time domain, assigning different processors to different regions of time.

6) Analytical Performance Analysis

. Felderman and Kleinrock(1990) showed that the average performance difference between synchronous time-stepping and an optimistic asynchronous algorithm such as Time Warp is no more than a factor of o(log P), P being the number of processors.

. Lin and Lazowska(1990) demonstrated conditions for the optimality of Time Warp(in the absence of overhead costs).

. Lipton and Mizell(1990) demonstrated an interesting asymmetry with examples showing that Time Warp

is capable of arbitrarily better performance than the Chandy-Misra-Bryant null-message approach and a proof that the converse is not true.

. Nicol(1991) studied the difference between a conservative windowing algorithm and Time Warp.

. Dickens and Reynolds(1991) analyzed a windowing algorithm.

. Gupta(1991) found a detailed analysis of Time Warp.

. Lin and Lazowska(1991) considered scheduling issues in Time Warp.

. Lin and Lazowska(1991) and Lubachevsky(1991) studied roll-back.

. Felderman and Kleinrock(1991 and 1992) studied that exact two-processor analyses permit a comparison of optimistic and conservative methods. However, this style of analysis is extended to general numbers of processors.

. Akyildiz(1992) considered an extension to consider the effects of limited memory.

Chapter 3. Implementing GPSS as a PDES with Roll-Back

1. Introduction

The obstacle to the implementation of PDES is synchronization conflicts. Thus, an implementation of PDES is focused on how to solve this problem. To be specific, this chapter shows the synchronization conflicts that can occur in GPSS(General Purpose Simulation System) by checking the statements of GPSS one by one. This chapter also presents the use of C-LINDA to implement some of GPSS as a PDES.

2. C-LINDA

(1) Overview[1,5,32,46]

LINDA is a general MIMD model of parallel computation based on distributed data structures. LINDA can be implemented for every parallel computer system of MIMD system and it can be embedded into different programming languages such as C, FORTRAN, MODULA-2, etc. C-LINDA is an implementation of the LINDA model using the C programming language. That is, C-LINDA is a parallel programming language based on C that enables users to create parallel programs.

C-LINDA has several advantages which set it apart from other parallel programming environments :

. C-LINDA augments the C programming language.

. C-LINDA parallel programs are portable. C-LINDA itself is available on a large number of parallel computer systems, including shared-memory computers, distributed memory computers, and networks, and with few exceptions, C-LINDA programs written for one machine run without change on another.

. C-LINDA is easy to use. C-LINDA implements parallelism via a logically global memory, called tuple space, and a small number of simple but powerful operations on it. In addition, the C-LINDA compiler supports all of the usual program development features, including compile-time error checking and runtime debugging and visualization.

Distributed data structure programs use a shared data space, which can be accessed by all processes simultaneously in order to read or write data. All interprocess communication is accomplished via this global data space. Processes never sent messages to one another, but rather place data into the shared data space. When one process needs that data, it obtains it from the shared data space and updates it if necessary.

In C-LINDA, programs use a master/worker computation strategy. Under this approach, the total work to be done by the program is broken into a number of discrete tasks which are stored in the global data space. One process, known as the master, is responsible for generating the tasks and gathering and processing the

results. Actual program execution involves a number of component processes known as workers. In this thesis, each worker executes its task independently until the master process determines it needs to do a rollback. The worker has to be backed up to the time just before the conflict. After roll-back, the worker executes again from this point in simulated time. This process is repeated until the program is totally finished.

(2) C-LINDA Operations[1,5,32,46]

C-LINDA calls the shared data space tuple space. The tuple space can be accessed by all processes. Data moves to and from the tuple space as tuples. Tuples can consist of multiple data elements of any type. A tuple is a sequence of up to 16 typed fields, called arguments; it is represented by a comma-separated list of items enclosed in parentheses. There are two kinds of tuples. One is data tuples, called passive tuples which contain static data and the other is process tuples, called live tuples or active tuples which are under active evaluation.

Six functions enable a process to interact with tuple space :

. An EVAL operation creates a live tuple process to evaluate each argument. It starts a new process. Once a passed tuple evaluated, it is placed in tuple space as an ordinary data tuple. Evaled functions may have a maximum of

16 parameters, and both their parameters and return values must be one of the following types : int, long, short, char, float, double. No structures, pointers, arrays, or unions are allowed as function arguments. Data of these types can always be passed to a process through tuple space.

. The IN operation attempts to read and remove a passive tuple from the tuple space by searching for a data tuple which matches the template specified as its argument.

. The INP operation is a in predicate that is a boolean test operation of the data in the tuple space. It tests whether a suitable data tuple exists. It returns a 1 if a matching tuple is retrieved, 0 if not.

. The RD operation attempts to read a passive tuple from the tuple space without erasing it. It does not remove the matching tuple from the tuple space.

. The RDP operation is a read predicate that is a boolean test operation of the data in the tuple space. It tests whether a suitable data tuple exists, without erasing it. It returns a 1 if a matching tuple is retrieved, 0 if not.

. The OUT operation is a generation of a passive tuple. It adds a tuple to tuple space. Prior to adding it, OUT evaluates all of its fields, resolving them to actual values. OUT returns after the tuple has been added to tuple space.

When executing a RD or IN operation, three different cases can arise :

. If there is exactly one matching tuple in the tuple space, then this tuple is read (or read and subsequently deleted (in operation)).

. If there are two or more matching tuples in the tuple space, then an arbitrarily matching tuple is read (or read and subsequently deleted (in operation)).

. If there is no matching tuple in the tuple space, then this process is blocked until a matching tuple becomes available (i.e., is written into tuple space by some other process using the out operation), or until the whole process system terminates.

In RD/IN operations, a variable may be used either as a value that has to match the corresponding component of a tuple in tuple space, or it may be used as a return variable, which is bound to take the component value from the tuple read. A variable used in the latter way has to be prefixed by a question mark "?", in order to distinguish these two cases.

(3) Tuples and Their Matching Rules

A tuple and a template match when[46] :

. They contain the same number of fields.

. All corresponding fields are of the same type.

. The type of a field containing an expression is whatever type the expression resolves to. The type of a field containing a formal is the type of the variable used in the

formal.

. For a structure or union field, the type is extended to include the structure or union name. The name and size of structures must match.

. The type of a pointer is the type of the object to which it points.

. Arrays match other arrays whose elements are of the same type. Thus, an array of integers will match only other arrays of integers and not arrays of characters.

. Scalar types don't match aggregate types.

. All corresponding fields are of the same size.

. Fixed length aggregate fields match only other fixed aggregates of the same length. In particular, fixed aggregates do not match varying aggregates even when they have the same length.

. The corresponding fields in the tuple contain the same values as the actuals in the template.

. Scalars must have exactly the same values. Care must be taken when using floating point values as actuals to avoid inequality due to round-off or truncation.

. Aggregate actuals such as arrays(which otherwise match) must agree in both the number of elements and the values of all corresponding elements.

3. Modeling for An Implementation

To implement PDES for GPSS, a program needs to be decomposed into several processes (LPs), referred to as workers, based on logical blocks of statements that start with a GENERATE statement. An additional LP referred to as a master is needed to start the workers, check for synchronization problems, and gather and report the results. The master process maintains a linked list ordered by time of the state of the system each time a statement that includes a global variable is executed. Each time a worker process executes a GPSS block in which a global variable is used a record is added to the linked list in the master process to check if a synchronization problem occurs. Every LP can be run on its own processor. Every worker process also maintains a linear list in ascending simulated time. That is, whenever an event is inserted into a linear list, it is inserted at the position determined by the simulated time. In order to make roll-back possible, the linked list must include a history of the worker process's own transaction movements and a future events list.

Interprocess communications between the master process and the worker process is by tuples in the tuple space. The tuple space is the shared data space in C-LINDA. A tuple, like the local time of the interrupt worker process, is stored in the tuple space to use when each worker process communicates with the master process for

checking the common passive resources and for the termination counter, if it is used by more than one worker process. Whenever a worker process requests some information from the tuple space, the process gets the data from the tuple space using the IN or RD function of C-LINDA. Likewise, whenever a process captures a common passive resource or terminates an event, the process stores the updated information into the tuple space using the OUT function of C-LINDA and then it must check whether the synchronization problem (out-of-order-execution) has occurred or not. If the out-of-order-execution has occurred, then all the executed events have to be rolled back to the time just before the conflict, using the ascending linear list in every worker process. Forward events in the linear list are destroyed or updated. Then local time is reset. If a synchronization problem has not occurred, each process continues to run without interruption.

Every LP contains the following components :

1) System state

An LP always may execute its currently next local event. In order to make roll-back possible, the system must save the state of event from time to time. To do this, every LP maintains an ascending queue by the simulated time using double linked list. That is, whenever an event is inserted into a queue it is inserted at the position determined by the simulated time and this queue contains

all states of the event to use when the roll-back is occurred. The states are as follows ;

- State variables : A set of variables describing a state.

- Simulation local clock : A variable giving the current value of simulated time.

- Event list : A list containing the next event time of each type.

- Statistical variables : A set of variables used for storing statistical information about system performance.

2) Initialization routine

A subprogram to initialize the system when the simulation starts.

3) Timing routine

A subprogram that advances the simulation local clock to the minimum next event time.

4) Event routine

A subprogram that determines the next event type from the event list.

5) Input routine

A subprogram that supports the input parameter if necessary.

6) Rollback routine

A subprogram to roll-back the effects of all prematurely executed events when synchronization conflicts occur.

7) Library routines

A set of subprograms used to generate random numbers.

The master LP maintains the following components:

1) Common-check routine

A subprogram to check synchronization problem of the common passive resource between LPs. When the synchronization problem occurs, the rollback routine in a worker LP is executed. Every worker LP maintains a linked list for the common passive resources in time order to use when backup is required.

2) Terminate-check routine

A subprogram to handle the synchronization problem of Termination Counter in START Statement in GPSS whenever every worker LP executes a TERMINATE statement. To check the exact termination time, every worker LP communicates with the master process using tuples in the tuple space.

The algorithm begins by the master process evaluating the worker processes. Every evaluated worker process executes its own program simultaneously. Each worker process runs to execute a task independently until it is interrupted by another worker process. When a worker process is interrupted by another worker process, the worker process has to be backed up to the time just before the conflict. After rollback, the worker process runs to execute again. This process is repeated until the program

is totally finished. After each worker process finishes its own program, each worker process returns the value to the master process. Then, the master process gathers the statistics and reports the results. The logical relationships (flow of control) between these components can be depicted as in Fig.8 and Fig.9. In Fig.9, □ represents a subprogram in every worker LP and ■ represents a subprogram in the master process.

4. Solutions of Synchronization Conflicts

In this section, the synchronization problems of GPSS statements are examined one by one. For every statement, shared variables have a temporal dimension, for instance a GPSS statement like TEST X A, B. Clearly the time at which a variable is executed has an effect an its value. But the value of a shared variable is not completely under the control of the process executing this statement.

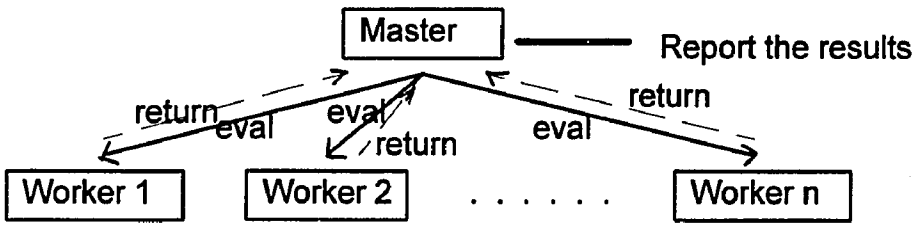


Fig.8. The relationship between the master and the workers

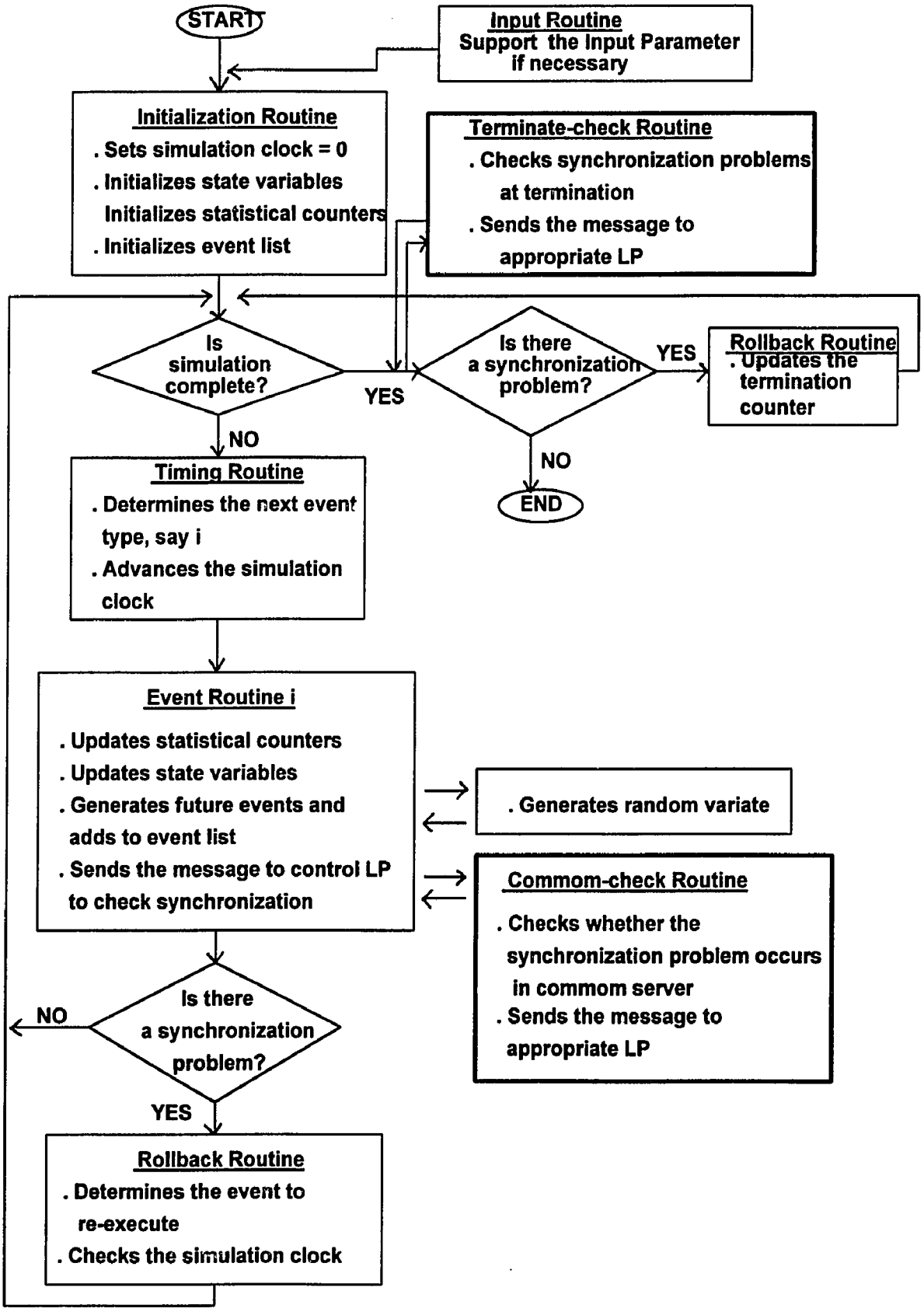


Fig.9. The relationship of the components in every worker

Thus if another process shares this variable, its value may not be determined at the time of execution of this statement. The best one can do to continue execution is to guess its value. The idea is to store a tuple with its time for later testing to see if the execution needs to be rolled back. If the value of the shared variable is changed by another process later, then the process has to be rolled back. In all examples, we suppose there are two worker processes, LP1 and LP2, and LP1 is ahead of LP2. Throughout we take the GPSS description of the statement from [37,38], and we present the GPSS statements in their basic form to simplify discussion. The same logic is also applied when we extend to more than two worker processes. If indirect addressing is included, the same basic structure still holds in the logic and the generalization is straight forward.

(1) GENERATE Statement : Creation of Event

The purpose of this statement is to create and schedule a next event. The general form is GENERATE A, B, C, D, E, where A is average interarrival time, B is half-range of the uniformly distributed interarrival time random variable, C is offset interval, D is limit count and E is event priority level.

In our implementation, for each GENERATE statement, the master LP initiates a new process using the C-LINDA eval operation.

(2) PRIORITY Block : Modification of a Transaction's Priority Level

When a transaction enters a model, its priority level is specified through the E operand at its GENERATE block. Whenever a transaction moves into the PRIORITY block, its priority level is changed. The general form is PRIORITY A, where A is the value to be assigned as the priority level of transactions which enter the PRIORITY block.

In our implementation, whenever a PRIORITY statement occurs, or a priority in the GENERATE statement is specified, priority value is stored in the transaction. If there are more than two transactions with the same time, then the transaction with the higher priority is first in order. Reordering may be required.

(3) ADVANCE Block (Providing for the Passage of Time)

When a transaction moves into an ADVANCE block, a simulated time advance is applied to the transaction. The ADVANCE block never refuses entry to a transaction. Any number of transactions can be held there simultaneously. Whenever a transaction moves into such a block, the underlying subroutine is executed again and a customized holding time is computed. The general form is ADVANCE A, B, where A is average service time and B is half-width of range over which holding time is uniformly distributed.

In our implementation, there is no difference from the sequential simulation.

(4) SEIZE/RELEASE statement (Handling of Common Servers)

The Facility entity (that is, Facilities) can be used in GPSS to model single servers. The purpose of the statement is to model individual servers. The general form is SEIZE/RELEASE A, where A is the identifier for the facility being captured (at the SEIZE) or being given up (at the RELEASE).

In our implementation, whenever a SEIZE/RELEASE statement occurs in a GPSS program and common server resource exists between LPs, synchronization conflicts must be checked without fail. If a synchronization conflict has occurred, then the effects of all executed transactions that have been executed with incorrect model logic have to be undone (rolledback). However, if no common server exists between LPs that have SEIZE/RELEASE statements, then we don't need to consider synchronization conflicts any further because LPs do not influence each other.

The algorithm to solve synchronization conflicts is as follows:

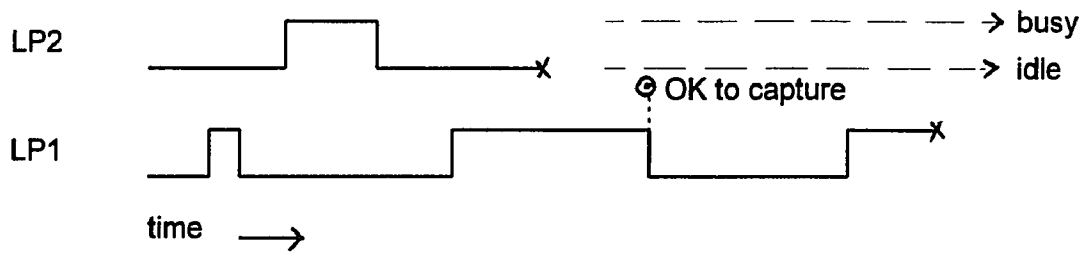
IF (LP2 is the executing process)

THEN {

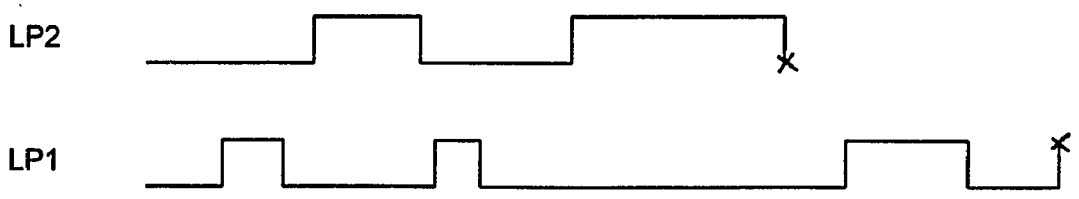
IF (LP1 has already captured the common

```
                server when LP2 requests it)
case1.          THEN  LP2 must wait until LP1 releases the
                      server
                ELSE {
                     LP2 captures the common server
                     IF (the time that LP2 releases the
                        server < the time that LP1 captures
                        the  next server)
case2.                  THEN  LP1 and LP2 continue to run : no
                              problem here
case3.                  ELSE  LP1 must be backed up until the
                              time that LP2 releases the server
                }
          }
          ELSE  LP1 and LP2 continue to run
```

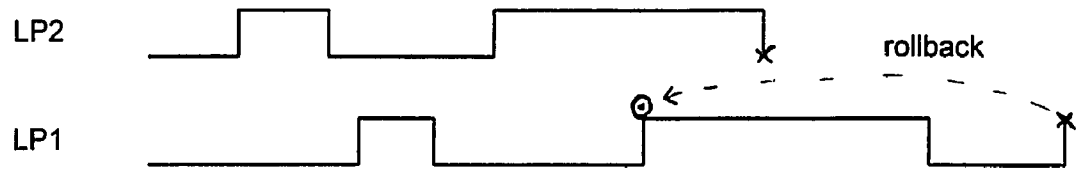
(case 1)



(case 2)



(case 3)



(5) STORAGE Control Statement : Specifying Group Size

The number of servers modeled with a particular storage, referred to as the storage's capacity, is indicated by including a STORAGE control statement for that storage. The general form is Label STORAGE A, where the label is a identifier for the storage whose capacity is being defined and A is the capacity of the storage.

In our implementation, if the identifier is global, then the identifier is stored with the storage value A.

(6) ENTER/LEAVE statement

This statement is a storage entity. The Storage entity (that is, storages) can be used to model groups of two or more servers who (or which) are indistinguishable from each other as servers. The general form is ENTER/LEAVE A, B, where A is the identifier for a storage (one of whose servers is being requested/captured at the ENTER, or being given up at the LEAVE) and B is the number of servers being

requested/captured at the ENTER, or being given up at the LEAVE. And the general form is Label STORAGE A, where Label is the identifier for the storage whose capacity is being defined and A is the capacity of the storage.

In our implementation, whenever a ENTER/LEAVE statement occurs in a GPSS program and common server exists between LPs, synchronization conflicts must be checked without fail. In this case, a record of each change in the storage's value is maintained in order according to its time.

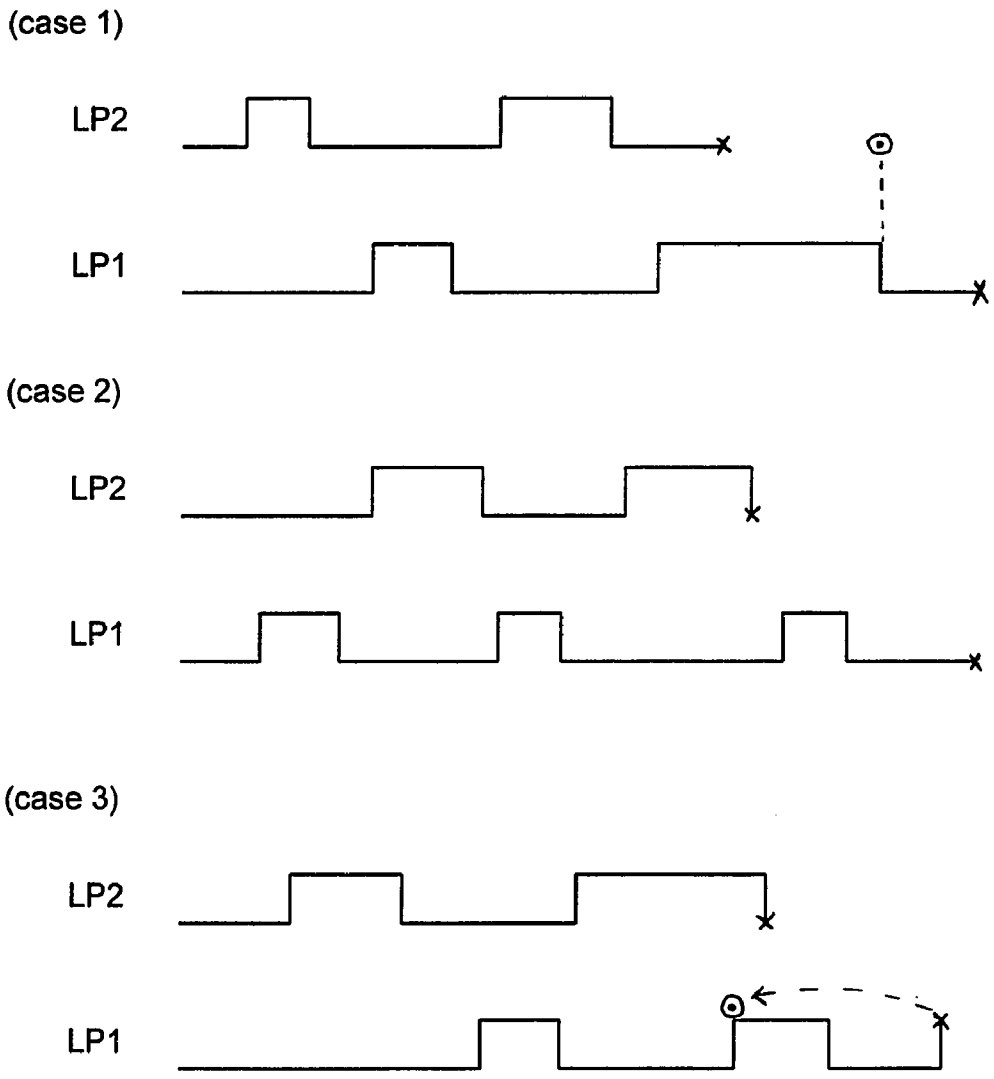
The algorithm to solve synchronization conflicts is as follows:

```
IF  (LP2 is the executing process)
THEN {
     IF (LP1 has already captured the common
        server when LP2 requests it)
     THEN {
          IF (the number remaining in storage of
             LP1 > the number in using storage of
             LP2)
          THEN  the number remaining in storage
                of LP2 = the number remaining in
                storage of LP1 - the number in
                using storage of LP2
case1.    ELSE  LP2 must wait until LP1 releases
                the storage
```

```
                }
                ELSE {
                     LP2 captures the storage
                     IF (the time that LP2 releases the
                        storage < the time that LP1 captures
                        the next storage)
case2.               THEN  LP1 and LP2 continue to run : no
                           problem here
                     ELSE {
                          IF (the number remaining in the
                             storage of LP2 > the number in
                              using storage of  LP1)
                          THEN  the number remaining in
                                storage of LP1 = the number
                                remaining in storage of LP2 -
                                the number in using storage
                                of LP1
case3.                    ELSE  LP1 must be backed up until
                                the time that LP2 releases
                                the use of the storage
                     }
             }
          }
          ELSE {
               IF (the number remaining in storage of LP1
                  > the number in using storage of LP1)
               THEN  LP1 and LP2 continue to run
```

ELSE LP1 must wait until LP1 release the storage

}



(7) QUEUE/DEPART Blocks : Gathering Statistics When Waiting Occurs

GPSS provides an option for automatically gathering statistics describing the involuntary waiting which may occur from time to time at various points in a model. A transaction starts its membership in a queue by executing a QUEUE Block, and then later ends its membership in the queue by executing a DEPART block. The general form is QUEUE/DEPART A, B, where A is the identifier of the Queue to be joined(QUEUE) or departed (DEPART) and B is the number of units by which the recorded content of the queue is to be modified.

In our implementation, whenever a QUEUE statement occurs in a GPSS program and the identifier is common between worker LPs, identifier A is stored in the tuple space with its time and some statistics in time order. The algorithm is as follows :

IF (A is local)
THEN there is no concern
ELSE save the queue statistics of this
tuple in time order

(8) START Control Statement

When a START Control statement is executed, the model's Termination Counter is given an initial value, the model's GENERATE blocks are initialized and block-to-block movement of transactions begins.

The general form is START A, where A is the initial value of the model's Termination Counter. The A operand must have a value of 1 or more.

In our implementation, value A is used like a constant. The value A is stored in the tuple space to use for each TERMINATE A(A > 0) statement.

(9) TERMINATE statement : Termination of Event

TERMINATE statements are used to destroy events(remove them from a model) and to provide the modeler with a tool for controlling the duration of a simulation.

The general form is TERMINATE A, where A is a decrement number for the model's Termination Counter. That is, a TERMINATE statement's A Operand indicates the amount by which the value of a counter in the model (called the model's Termination Counter, or TC for short) is to be reduced each time a event destroys itself by moving into the TERMINATE statement.

When the value of the Termination Counter has been reduced to zero (or less), the simulation stops immediately. A model can contain any number of TERMINATE statements, but the model has only one Termination Counter.

In our implementation process, if only one TERMINATE A (A > 0)statement exists in GPSS program, then we don't need to consider synchronization conflicts because they will not occur. But, if two and more TERMINATE A (A >

0) statements exist in a program, synchronization conflicts must be checked without fail. Whenever an LP executes a TERMINATE A (A > 0) statement, the LP stores the information including the simulation clock time and the value of its own Termination Counter into the tuple space and checks whether the synchronization problem has occurred or not.

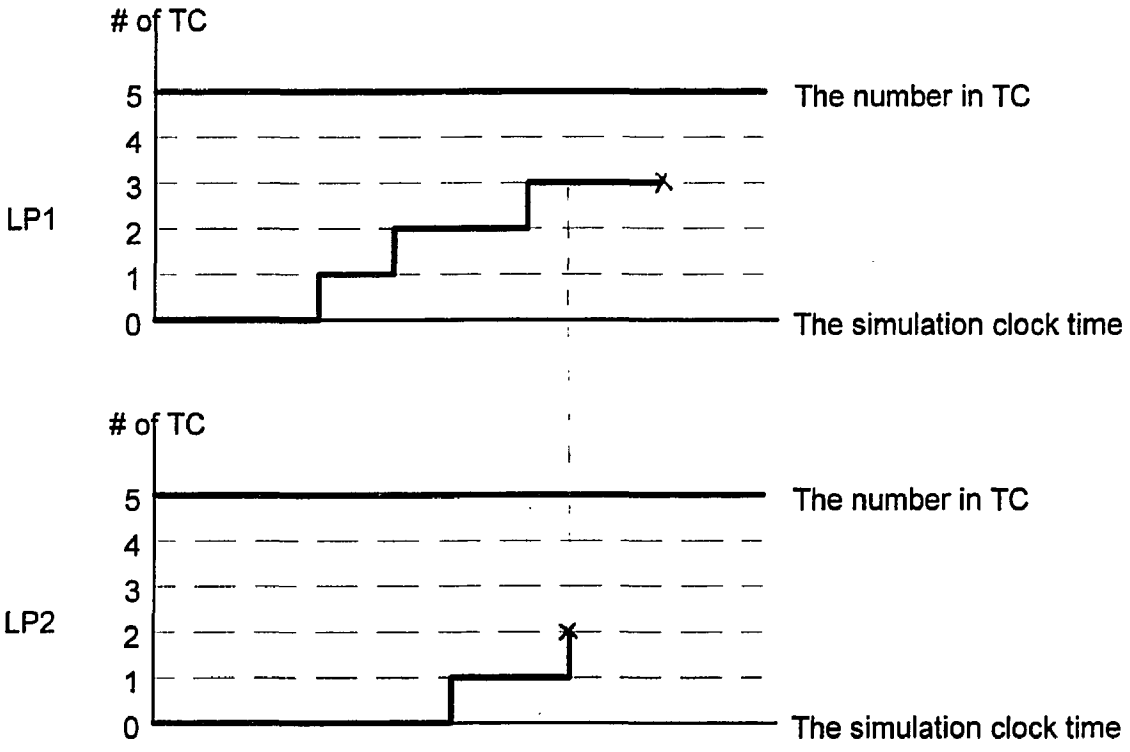
The algorithm to solve synchronization conflicts is as follows :

```
IF (LP1 is the executing process)
THEN  the number in TC of LP1 is saved and
      LPs continue to run
ELSE {
      . find the right place for the number in
        TC of LP2 by its time
      IF (the number in TC of LP1 + the number in
         TC of LP2 >= the number in TC)
      THEN {
           IF (the number in TC of LP1 + the
              number in TC of LP2 == the number
              in TC)
Case1.     THEN  all LPs stop to run with the
                 simulation clock time of LP2
Case2.     ELSE  all LPs stop to run with the
                 simulation clock time of LP1
      }
```

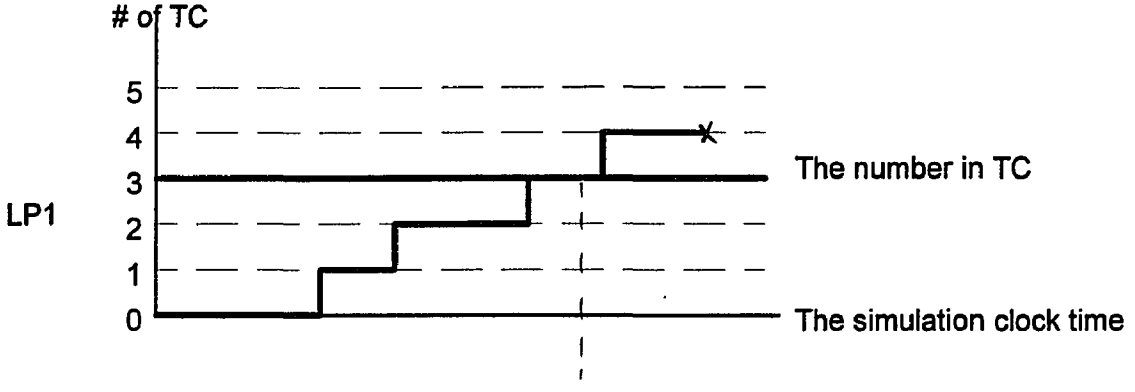
Case3. **ELSE** the number in TC of LP2 is saved
and LPs continue to run

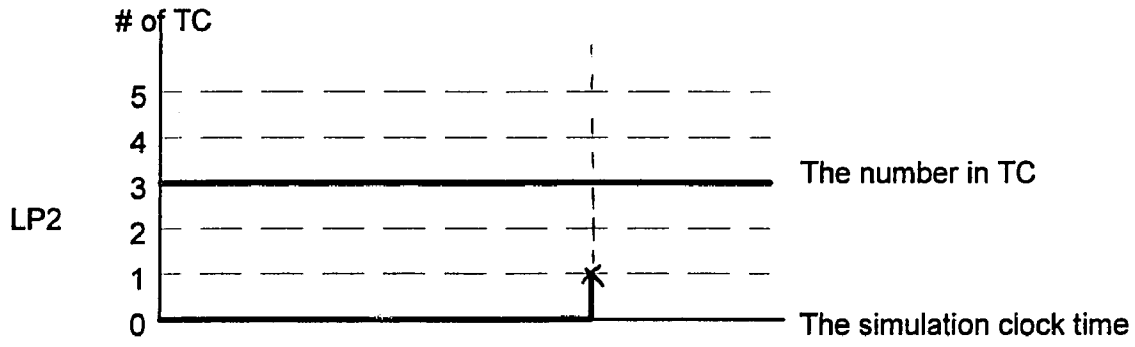
}

(case 1)

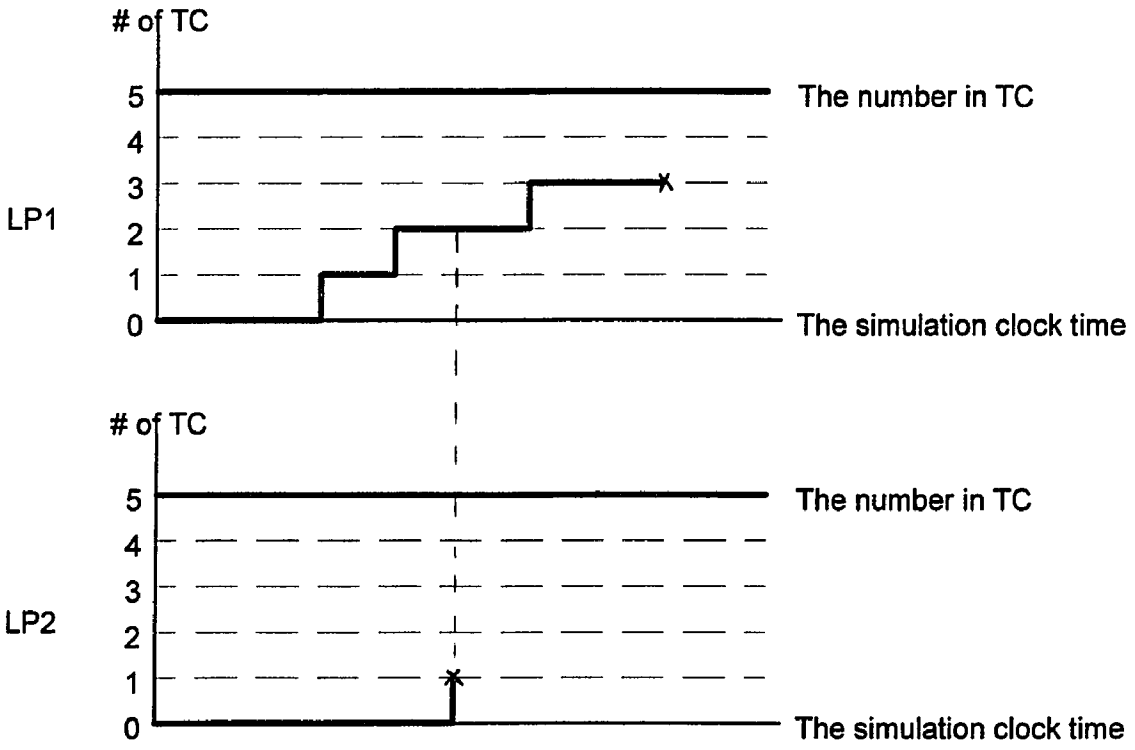


(case 2)





(case 3)



(11) TEST Block : Testing Numeric Relationships

The relation between the values of two standard numerical attributes can be examined by use of the TEST block. The general form is TEST X A,B[,C], where A is the name of the first standard numerical attribute, B is the

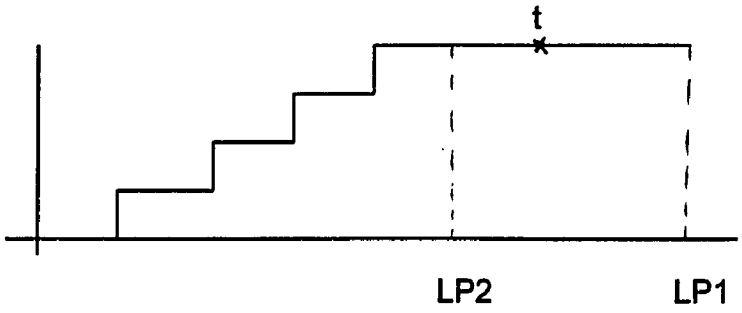
name of the second standard numerical attribute, X is the auxiliary operator which represents the relational operator to be used in the test; the forms X can assume are G, GE, E, NE, LE, and L, and C is a operational operand; block location to which the testing transaction moves if the answer to the question implied by the relational operator is "no". When the TEST block's C operand is used, the test is conducted in conditional transfer mode.

In our implementation, whenever the program arrives at the TEST Block, the algorithm is as follows :

```
IF (both  A and B are local variables or one is
   constant and the other is local)
THEN  there is no concern
ELSE  {
     IF (LP1 is the executing process)
     THEN {
          IF (C is not used)
          THEN {
               REPEAT {
               . find the right place for the
                 variables A and B by their time
               . use the previous value of the
                 variables A and B as the values
                 of A and B
               . execute the TEST statement
               }
```

```
                    UNTIL (the TEST statement is true)
                }
                ELSE {
                    IF (the TEST statement is false)
                    THEN  goto the statement C
                }
Case1.              . save the "TEST" state in the linked
                      list
            }
             ELSE {
                . find the right place for the
                 variables A and B by their time
                . use the previous value of the
                 variables A and B as the values of A
                 and B
                . execute the TEST statement
             }
        }
```

(case 1)



(11) INITIAL, SAVEVALUE Block

Normally, the GPSS processor sets savevalues to zero before a simulation begins. Selected savevalues can be initialized with nonzero values by use of the INITIAL card. The general form is INITIAL name1,value1/.../namei,valuei/.../namen,valuen.

The value of one savevalue is modified when a transaction moves into a SAVEVALUE block. The general form is SAVEVALUE A,B,C, where A is the number or symbolic name of the savevalue to be modified, B is data to be used in the modification process, and C specifies whether the savevalue involved is a halfword or fullword type. SAVEVALUE block can be used in increment mode and decrement mode, as well as in replacement mode. In increment mode, the previous value of the savevalue is incremented by the B operand data. In decrement mode, it is decremented by the B operand data. Increment and decrement mode are specified by placing a plus or minus sign, respectively, ahead of the comma which separates the A and B operands.

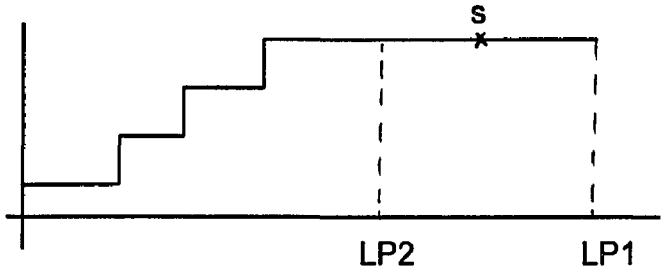
In our implementation, when the simulation begins and the identifier is global, its value is stored in the tuple space. Whenever a SAVEVALUE is executed of a common identifier, synchronization conflicts must be checked without fail. The algorithm is as follows :

IF (A and B are local)

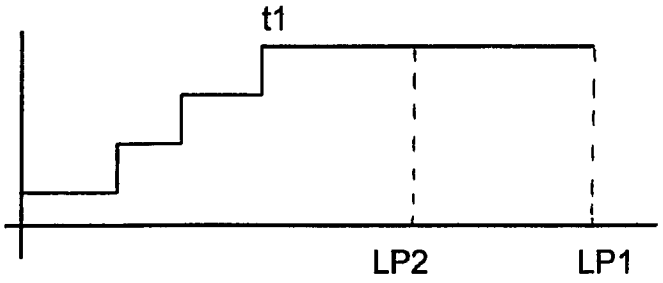
THEN there is no concern

```
        ELSE  {
              . find the right place for the global
                variables by their time
              . read the previous value of the global
                variable
              IF (LP1 is the executing process)
Case1.        THEN  save the values of variables A and B
                    with its time and the mark s
              ELSE  {
                    . t1 = the time of the last change of A
                      in LP1
                    If (t1 < the local clock time of LP2)
Case2.              THEN  continue to run
Case3.              ELSE  LP1 must be backed up to the
                          local clock of LP2
              }
        }
```

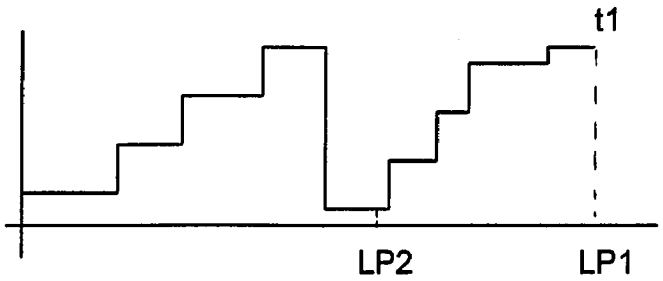
(case 1)



(case 2)



(case 3)



(12) ASSIGN Block : Modification of Parameter Values

The general form is ASSIGN A, B, C, where A is the number of the parameter to be modified, B is the data to be used for the modification and C is the number of a function. When a transaction enters an ASSIGN Block, the B-Operand data is copied to the parameter whose number is provided by the A operand. In replacement mode, the old value of a parameter is replaced with a new value, without regard to what the old value might have been. In increment mode, the parameter's new value is computed by adding the B operand data to the old value. In decrement mode, the

parameter's new value is computed by subtracting the B operand data from the old value. Increment and decrement mode are specified by placing a plus or minus sign, respectively, ahead of the comma separating the A and B operands.

The processor performs these steps in executing the ASSIGN block subroutine when the C operand is used.

. The C operand is evaluated.

. The function whose number equals the C operand's value is evaluated.

. The function's entire value is combined multiplicatively with the B-Operand data.

. The integer portion of the product is used to replace, increment, or decrement the parameter specified with the A operand.

In our implementation, whenever an ASSIGN statement occurs in a GPSS program, the algorithm is as follows :

```
IF (A, B and C are local)
THEN there is no concern
ELSE  {
      . find the right place for the global
       variables by their time
      . read the previous value of the
        global variables
     IF  (LP1 is the executing process)
```

```
            THEN  save the values of variables A, B
                  and C with its time and the mark t
            ELSE  {
                  . t1 = the time of the last change of A
                   in LP1
                  IF (t1 < the local clock time of LP2)
                  THEN  save the values of variables
                        A, B and C with its time and
                        continue to run
                  ELSE  LP1 must be backed up to the
                        local time of LP2
            }
      }
```

(12) SELECT Block : Finding Entities Satisfying Stated Conditions

GPSS provides a block which can be used to scan a specified set of entity members to determine if at least one of the members currently satisfies a stated numeric condition. When a transaction moves into the block, it triggers a single scan over the specified entity members. The members are scanned in order of increasing number. If a member is found which satisfies the stated condition, the scan immediately terminates. Otherwise, the scan terminates when it has been determined that no members in the specified set currently satisfy the stated condition.

Either way, the selecting transaction moves on in the model when the scan terminates. The general form is SELECT X, A, B, C, D, E, F, where A is the number of the parameter into which the number of an entry member currently satisfying the stated condition is to be copied, B and C are the smallest and largest numbers, respectively, in the set of entity members subject to the scan, D is the data against which the E-Operand SNA is to be compared, E is the family name of the Standard Numerical Attribute being investigated, F is the optional operand; block location to which the selecting transaction moves if no entity member currently satisfies the indicated condition, and X is an auxiliary operator;it represents the relational operator which specifies the way in which the E-Operand SNA is to be compared to the D-Operand data; X takes one of the forms G, GE, E, NE, LE, and L.

In our implementation, this statement is almost the same as the ASSIGN statement except that the global variables are from Eb to Ec instead of B and C. In the ASSIGN statement, if the statement is true then the next statement is executed. In the SELECT statement, if the statement is false then F statement is executed as the TEST statement.

(13) TABLE Card and TABULATE Block : Defining and Using Tables

Use of the table entity involves two steps. First, each of the one or more tables to be used in a model must be defined. Second, arrangements must be made to have sampled values entered, one by one, in the various tables of interest. The general form of TABLE is TABLE A, B, C, D, where A is the name of the random variable whose values are to be entered in the table, B is the first boundary point, C is the width of each intermediate table interval, and D is the total number of intervals in the table, including the leftmost and rightmost.

Sampled values are entered into a table one by one, as a simulation proceeds. A value is entered into a table each time a transaction moves into a TABULATE block. The general form is TABULATE A, where A is the name of the table into which a value is to be entered.

In our implementation, whenever the program executes a TABULATE block, the entry of the value A is increased in its table. The algorithm is as follows :

IF (A is local)
THEN there is no concern
ELSE the value is stored in time order
like QUEUE

(14) LOGIC Block

The status of a logic switch can be changed by

having a transaction enter the LOGIC block. The potentially changed position of the logic switch may make it possible for one or more previously blocked transactions in a model to resume their movement. The general form is LOGIC X A, where A is the name of a logic switch and X is the auxiliary operator which indicates what is to be done to the indicated logic switch; the forms X can assume are R, S, and I.

In our implementation, whenever the program arrives at the LOGIC block, the algorithm is as follows :

```
IF  (A is local)
THEN  there is no concern
ELSE  {
      . find the right place of global variable A
       by its time
      . read the previous value of A
      IF  (LP1 is the executing process)
      THEN  save the changed value of A
            with its time
      ELSE  {
            . save the changed value of A  with
            its time
            . t1 = the time of the last change of
            A in LP1
            IF (t1 < the local clock time of LP2)
            THEN  keep going to execute
```

ELSE LP1 must be backed up to the
local clock of LP2
}
}

(15) GATE Block : Testing the Setting of Logic Switches

When the setting of a logic switch is tested, no numeric properties of the switch are involved. In fact, a switch has no numeric properties. This suggests that some block other than the TEST Block must be introduced to control the flow of transactions as a function of the set or reset status of logic switches. The block used for this purpose is the GATE block. The general form is GATE X A, B, where A is the name of a logic switch, B is the optional operand; block location to which the testing transaction moves if the logic switch is not in the condition required for the test to be true, and X is the auxiliary operator, termed a logical mnemonic, indicating the switch setting which is required for the test to be true; the two logical mnemonics for logic switches are LS and LR.

When the GATE block's B operand is used, testing at the GATE is conducted in conditional transfer mode. A transaction which arrives at the GATE moves to the sequential block if it is true that the logic switch has the indicated setting; otherwise, it moves to the nonsequential location indicated via the B operand.

In our implementation, whenever the program arrives at the GATE block, the algorithm is as follows :

```
IF  (A is local)
THEN  there is no concern
ELSE {
     IF (LP1 is the executing process)
     THEN {
          IF (B is not used)
          THEN {
               REPEAT {
               . find the right place of the
                global variable A by its time
               . use the previous value of the
                 variable A as the value of A
               . execute the GATE statement
               }
               UNTIL (the GATE statement is true)
          }
          ELSE {
               IF (the GATE statement is false)
               THEN goto the statement B
          }
          . save the state "GATE" in the linked
           list
     }
     ELSE {
```

```
                . find the right place for the variable
                  A by its time
                . use the previous value of the
                 variable A as the value of A
                . execute the GATE statement
            }
        }
```

(16) MARK Block

When a transaction enters the MARK block, the absolute clock's value is copied into one of its parameters. The general form is MARK A, where A is the number of the parameter into which the absolute clock's value is to be copied.

In our implementation, this statement is almost same as the ASSIGN A, B statement where B represents the local clock time of the program that executes the MARK block.

(17) TRANSFER Block : Nonsequential Movement of Transactions in a Model

It is occasionally of interest to divert transactions unconditionally to some nonsequential block in a GPSS model. This can be accomplished by using the TRANSFER block in unconditional transfer mode. The general

form is TRANSFER B,C where B and C are the block location to which transactions move next.

In our implementation, if B and/or C statement are in the same program, it is executed as a sequential simulation. But, if B and/or C statement are in the different program, the statement is not implemented in the parallel program.

IV. Experiments Involving the Implementation of GPSS as a PDES using C-LINDA

In this chapter, we use a C-LINDA program to experimentally examine the speed of a simple GPSS program that involves interactions between processes. It is felt that this type of interaction is general enough to illustrate some of the difficulties of using PDES for GPSS programs.

For convenience, we define a Gprocess as a logical block of GPSS statements whose first statement is a GENERATE block. The block consists of those statements that a transaction will go through either until it is terminated or loops through repeatedly. We assume that transactions do not transfer from one Gproecss to another. We do allow logical interactions between Gprocesses. Thus a LOGIC block's test condition can depend on a transaction in another Gprocess or a facility can be in use because a transaction in another Gprocess is using it. In fact, it is these types of interactions that can cause rollbacks. From now on we use the term Gprocess instead of worker process as defined by C-LINDA.

1. Statement of the tugboat problem

This problem comes from Schriber[38]. Type A and Type B ships come to a small harbor to unload cargo. The

harbor consists of two berths. One of these berths is used only by Type A ships. The other berth is used only by Type B ships. There is one tugboat at the harbor. The tugboat is used to tug a ship into the berth prior to unloading, and then later to tug the ship back out of the berth after unloading has been completed. The services of the tugboat are not needed while a ship is unloading.

Both ship types compete for use of the one tugboat at the harbor. Service order for tugboat use, and for use of the A and B berths, is FCFS.

When a ship arrives at the harbor, it does these things:

. It requests the berth of the type it uses.
. After capturing the berth, it requests the tugboat.
. After capturing the tugboat, it gets pulled into the berth.
. It gives up the tugboat.
. It unloads cargo.
. It requests the tugboat again.
. It gets pulled out of the berth.
. It gives up the tugboat.
. It gives up the berth.
. It leaves the harbor.

The interarrival-time and unloading-time distributions for Type A ships are 15 +(-) 5 hours and 8 +(-) 2 hours, respectively. The respective interarrival-time and unloading-time distributions for Type B ships

are 20 +(-) 5 hours and 10 +(-) 4 hours. Berthing and deberthing times are 2 and 1 hours, respectively, independent of ship type.

2. Program

(1) GPSS tugboat program

```
     SIMULATE              base time unit:1hour
* Gprocess1
     GENERATE  15,5        Type A ships arrive, one by one
     SEIZE     BERTHA      request/capture the A berth
     SEIZE     TUGBOAT     request/capture the tugboat
     ADVANCE   2           berthing time
     RELEASE   TUGBOAT     let the tugboat go
     ADVANCE   8,2         unloading time
     SEIZE     TUGBOAT     request/capture the tugboat again
     ADVANCE   1           deberthing time
     RELEASE   TUGBOAT     let the tugboat go
     RELEASE   BERTHA      no longer occupying the A berth
     TERMINATE 1           Type A ships leave, one by one
* Gprocess2
     GENERATE  20,5        Type B ships arrive, one by one
     SEIZE     BERTHB      request/capture the B berth
     SEIZE     TUGBOAT     request/capture the tugboat
     ADVANCE   2           berthing time
     RELEASE   TUGBOAT     let the tugboat go
```

```
     ADVANCE   10,4       unloading time
     SEIZE     TUGBOAT    request/capture the tugboat again
     ADVANCE   1          deberthing time
     RELEASE   TUGBOAT    let the tugboat go
     RELEASE   BERTHB     no longer occupying the B berth
     TERMINATE 1          Type B ships leave, one by one
*
     START     500        start the Xact-Movement Phase
     END                  end of Model-File execution
```

(2) A Parallel program implementing parallel GPSS-like code using C-LINDA

We implemented a GPSS program and our logic for checking and correcting synchronization problem using C-LINDA. The program included GENERATE, SEIZE/RELEASE and TERMINATE statements. In the shown GPSS program, there are two logical blocks that start with a GENERATE statement. So, a master process and two Gprocesses are needed. A master process maintains a linked list ordered by time of the state of the system each time a statement that includes a global variable is executed. Every time a Gprocess executes a GPSS block in which a global variable is used a record is added to the linked list in the master process to check if a synchronization problem occurs. Every Gprocess also maintains a linked list to use if a rollback is needed. The linked list is a history of its own transaction

movements and a future events list. A tuple, like the local time of the interrupt Gprocess, is stored in the tuple space to use when each Gprocess communicates with another for checking the common passive resource tugboat and the termination counter. The master process starts Gprocesses which then execute simultaneously.

Each Gprocess first execute a initialize routine as in Fig.10. In the initialize routine, the simulation local clock, state variables, and statistical counters are initialized. In the timing routine, the event type of the next event to occur is determined. Every Gprocess has two kinds of events. One is an arrive event for the GENERATE statement in the GPSS program and the other is a demand event for the ADVANCE statements in the GPSS program. Whenever a Gprocess wants to use the common passive resource, e.g. the tugboat, the check-tugboat routine is executed. In the check-tugboat routine, the Gprocess interrupts the master process by sending a message with its local time and the tugboat duration time to the master process using the tuple space to check if a synchronization problem has occurred. In the seize routine in the master process, the logic of SEIZE/RELEASE statement is used. In this routine, a synchronization problem sometimes occurs. To check the synchronization problem, the master process finds the transaction's right place by the local clock time of the interrupt Gprocess in the linked list of the master process and then compares the interrupt tugboat duration

time with the tugboat duration time of the adjacent lists. If a synchronization problem does not occur, then the master process sends a message to the interrupt Gprocess using the tuple space and the interrupt Gprocess continues to run. If a synchronization problem occurs, then the

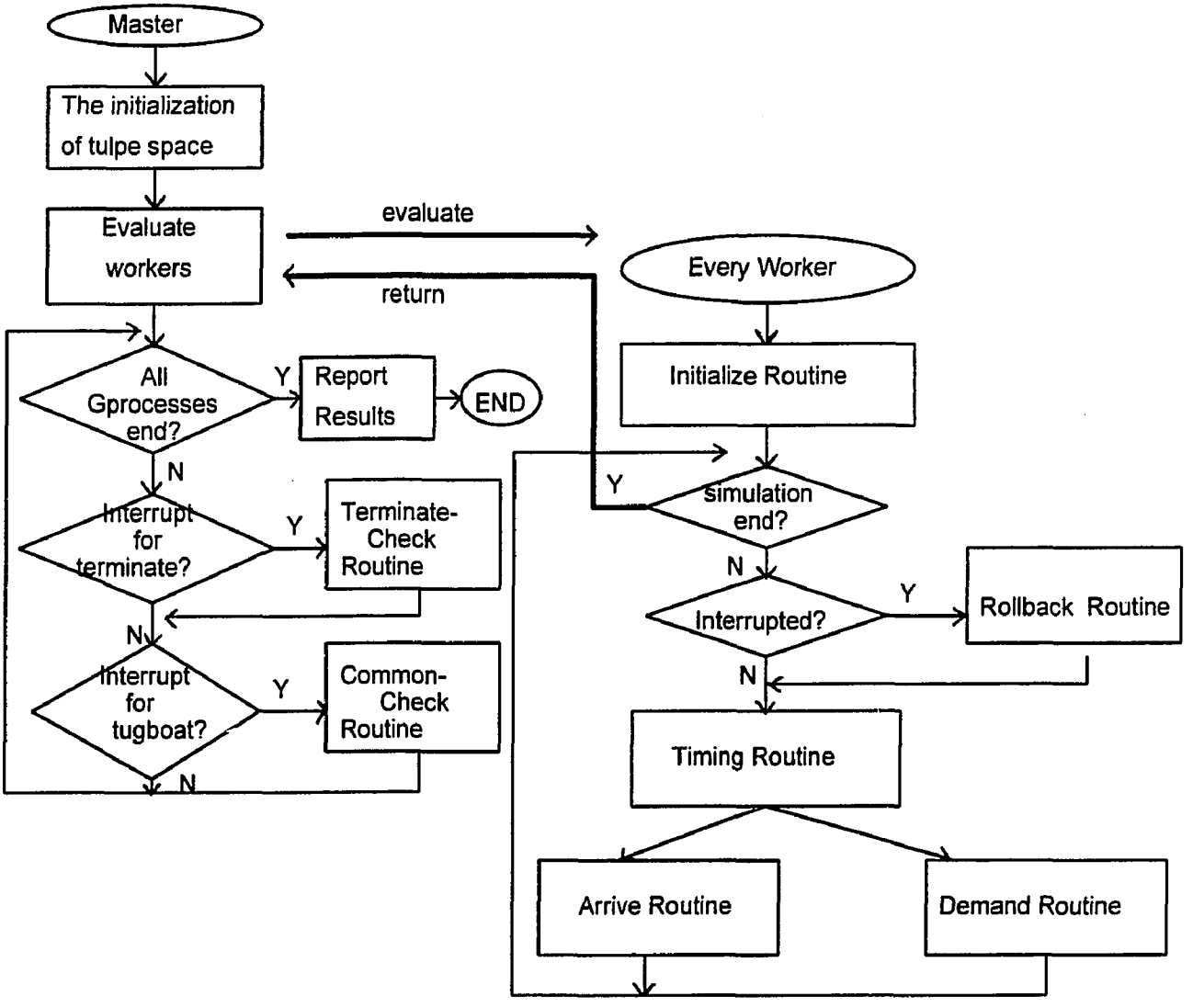


Fig.10. The flowchart of the problem

process interrupts another Gprocess. The interrupted Gprocess searches its own linked list to find the right place by the interrupted time using the tuple space and then the rollback routine is executed. In the rollback routine, the transactions of the interrupted Gprocess are deleted and the number of rolled back transactions are increased. Then the number of rollbacks are increased and the transactions of the interrupted process are deleted. Whenever a ship terminates, the depart routine is executed. In this routine, the Gprocess interrupts the master process by sending a message with its time and its termination counter to the master process using the tuple space to check the Termination Counter. In the terminate-check routine in the master process, the logic of the TERMINATE statement is used. In this routine, to check the synchronization problem for the Termination Counter, the master process finds its right place by the local clock time of the interrupt process. If the sum of the termination counter of the preceding Gprocess in the linked list and the termination counter of the interrupt Gprocess is smaller than the expected termination counter, then the master process sends a message indicating there is no synchronization problem to the interrupt Gprocess using the tuple space and the Gprocess continues to run. But, if the sum is greater than or equal to the expected termination counter, then all Gprocesses stop to simulate with the exact time.

After each Gprocess finishes, it returns its values to the tuple space. After all Gprocesses complete, the master process gathers the statistics and reports the results.

3. Results and discussion

(1) GPSS program results

The theoretical utilization is as follows :

Berth 1 utilization = 11 / 15 = 0.733

Berth 2 Utilization = 13 / 20 = 0.650

Tugboat Utilization = 3 / 15 + 3 / 20 = 0.350

The following table shows the results of the GPSS version. In this table, all of the utilization at TC = 500 are very close to the theoretical utilization. So, we use the number 500 as a termination counter in the parallel program.

	GPSS/H		
	TC=50	TC=100	TC=500
simulation time	425.947	865.364	4288.950
berth1 utilization	0.841	0.778	0.753
berth2 utilization	0.592	0.640	0.672
tugboat utilization	0.357	0.349	0.350

(2) Results of the parallel program using C-LINDA

We ran the C-LINDA version of the GPSS tugboat program. The execution times with one processor and the execution times with three processors are compared using a termination count of 50, 100 and 500. The results of ten repeated runs are shown. The results of TC=50, TC=100, and TC=500 with one processor are shown in Fig.11; the table in Fig.12 compares the execution time of one processor vs. three processors.

In the table, "CDS" means the master process and the Gprocesses are all executed on a single processor, and "NETWORK" means the master process and each Gprocess executes on different processor. "ORIGINAL" means the berthing and deberthing times of tugboats are as in the GPSS program. In both "CHANGE1" and "CHANGE2", the berthing and deberthing times of the tugboats are changed to 0.0 in Gprocess 2. In this way, the variable tugboat of Gprocess 2 is needed for zero time, consequently there will not be a synchronization problem between the processes and there will not need to be rollback. The difference between "CHANGE1" and "CHANGE2" is that "CHANGE1" doesn't check the synchronization problem for a tugboat, whereas "CHANGE2" does check using the tuple space whenever Gprocess 2 executes the SEIZE/RELEASE TUGBOAT statement, even though the synchronization problem does not occur. In this way, "CHANGE2" can be used to estimate the search time for checking the synchronization problem.

C-LINDA (CDS)	TC=50	TC=100	TC=500
simulation time	442.410	876.459	4338.349
execution time	3 (3, 4, 4, 4, 4, 4, 4, 4 5, 4)	5 (6, 4, 6, 5, 5, 5, 4, 6 4, 5)	23 (23,23,24, 21,24,22,22, 25,23,22)
berth 1 utilization	0.711	0.725	0.731
berth 2 utilization	0.604	0.632	0.639
tugboat utilization	0.342	0.342	0.348
# of rollback	13 (7,17,12,17, 14,14,15,12, 12,13)	28 (23,18,39,31, 24,22,25,42, 28,26)	126 (136,122,120 113,124,128, 155,110,135, 120)
# of rolled back tran.	135 (110,117,72, 140,92,87, 146,99,237, 252)	235 (236,141, 320,308, 186,164, 174,411, 236,174)	1351 (1066,1681, 897,948, 1147,1192, 2092,1166, 1444,1873)

Fig.11. The execution on one processor

From these cases, we learned that :

. When the "CHANGE1" was executed using one processor, at TC = 500, the execution time was 12 seconds. This large amount of time for such a simple program shows that C-LINDA has a lot of overhead.

. When the "CHANGE2" program was executed with one processor, at TC = 500, the execution time increased to 17 seconds, an increase of 5 seconds over the execution time of "CHANGE1". In the "CHANGE2" program, when TC = 500, the

		TC = 50	TC = 100	TC = 500
CDS	ORIGINAL	3 (2, 3, 3, 3, 3, 3, 3, 2, 4, 2)	5 (6, 4, 6, 5 , 5, 5, 4, 6, 4, 5)	23 (23,23,24,21,24, 22,22,25,23,22)
	# of rollback	13 (7,17,12,17,14, 14,15,12,12,13)	28 (23,18,39,31,24, 22,25,42,28,26)	126 (136,122,120,113, 124,128,155,110, 135,120)
	# of rolledback transaction	135 (110,117,72,140, (92,87,146,99, 237,252)	235 (236,141,320,308, 186,164,174,411, 236,174)	1351 (1066,1681,897,948, 1147,1192,2092, 1166,1444,1873)
	CHANGE1	2 (1, 2, 1, 2, 2, 1, 2, 1, 2, 2)	3 (2, 3, 3, 3, 4, 4, 3, 3, 2, 3)	12 (11, 16, 11,10,12, 13,14, 11, 11, 11)
	CHANGE2	3 (2, 3, 3, 3, 2, 2, 3, 3, 3, 2)	4 (4, 5, 4, 4, 5, 4, 4, 4, 4, 4)	17 (20, 16, 18, 15, 14, 21, 18, 17, 15, 17)
NETWORK	ORIGINAL	2 (2, 2, 2, 2, 2, 2, 3, 2, 3, 2)	4 (5, 4, 4, 3, 6, 4, 3, 4, 4, 4)	18 (16, 20, 17, 21, 13, 22, 15, 17, 16, 19)
	# of rollback	14 (10, 12, 16, 18, 13, 18, 25, 9, 12, 11)	28 (24, 26, 35, 38, 17, 25, 31, 29, 30, 26)	130 (119,113,119,158, 140,138,127,119, 126,140)
	# of Rolledback Transaction	127 (103,106,146,152, 149,128,99,120, 150,118)	291 (165,307,400,220, 179,205,367,345, 422, 298)	1319 (1359,1254,1438, 1253,1388,887, 1361,1434,1376, 1440)
	CHANGE1	1 (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	2 (2, 2, 2, 2, 2, 2, 1, 2, 2, 2)	7 (8, 8, 6, 8, 8, 6, 6, 7, 7, 8)
	CHANGE2	2 (2, 1, 1, 1, 2, 2, 2, 2, 1, 1)	3 (3, 3, 3, 2, 2, 4, 3, 3, 3, 4)	12 (11,9, 14,17,10, 12,14,13,9,13)

Fig.12. The execution on one processor and three processors

number of searches to check the synchronization problem is 1000 because this program has two SEIZE/RELEASE TUGBOAT statements. So, we know that the search time per 1000 is about 5 seconds. When the "CHANGE2" program was executed with 3 different processors, the execution time increased to 12 seconds, an increase of 5 seconds over the execution time of "CHANGE1". Therefore, the search time per 1000 is also about 5 seconds or about 100 searches per 0.5 seconds. This is also consistent in all the executions of TC = 50 and TC = 100.

. When the "ORIGINAL" program was executed on one processor, at TC = 500, the execution time increased from 17 to 23 seconds as compared with the execution time of the "CHANGE2" program. Also when the "ORIGINAL" program was executed with three processors, at TC = 500, the execution time increased from 12 to 18 seconds. In both cases, the increase of about 6 seconds was because of rollback overhead that was about 125 rollbacks and the search time of about 1400 transactions from the rollback. This also shows that the search time per 1000 is about 5 seconds.

. The program was extended to three Gprocesses and to four Gprocesses as follows :

```
*     Gprocess3
          GENERATE  40
          SEIZE     BERTHC
          SEIZE     TUGBOAT
```

```
            ADVANCE   2
            RELEASE   TUGBOAT
            ADVANCE   10,4
            SEIZE     TUGBOAT
            ADVANCE   1
            RELEASE   TUGBOAT
            RELEASE   BERTHC
            TERMINATE 1
*     Gprocess4
            GENERATE  50,5
            SEIZE     BERTHD
            SEIZE     TUGBOAT
            ADVANCE   2
            RELEASE   TUGBOAT
            ADVANCE   20,5
            SEIZE     TUGBOAT
            ADVANCE   1
            RELEASE   TUGBOAT
            RELEASE   BERTHD
            TERMINATE 1
```

The theoretical utilization is as follows :

Berth 1 Utilization = 11 / 15 = 0.733

Berth 2 Utilization = 13 / 20 = 0.650

Berth 3 Utilization = 13 / 40 = 0.325

Berth 4 Utilization = 23 / 50 = 0.460

Tugboat Utilization = 3 / 15 + 3 / 20 = 0.35

(if 2 Gprocesses are executed)

= 3 / 15 + 3 / 20 + 3 / 40 = 0.425

(if 3 Gprocesses are executed)

= 3 / 15 + 3 / 20 + 3 / 40 + 3 / 50

= 0.485

(if 4 Gprocesses are executed)

The execution time of the extended program with one processor vs. several processors at TC = 500 is given in Fig.13.

In Fig.13, in the NETWORK version, 2 Gprocesses means that the program is executed by 3 processors, one is for the master process and two are for two Gprocesses. When a program was executed with three processors, the execution time decreased from 23 to 18 seconds that is a decrease of 5/23 = 0.217 as compared with the execution time on one processor. When a program was executed with four processors, the execution time decreased from 42 to 30 seconds, that is a decrease of 2/7 = 0.286. When a program was executed with 5 processors, the execution time decreased from 71 to 45 seconds, that is a decrease of 26/71 = 0.366. This proves that the decreasing ratio increases when the number of processors increase.

. We breaks down the rollbacks, the transactions rolledback, and the wasted simulation time because of rollback, as per each process in Fig.14.

		2 Gprocess	3 Gprocess	4 Gprocess
CDS	execution time	23 (23,23,24,21,24, 22,22,25,23,22)	42 (42,39,42,42,36, 46,46,36,52,39)	71 (83,70,58,58,58, 91,70,74,87,62)
	# of rollback	126 (136,122,120,113, 124,128,155,110, 135,120)	273 (293,247,257,216, 275,324,238,297, 365,216)	412 (480,334,333,365, 338,558,404,441, 432,426)
	# of rolledback transaction	1351 (1066,1681,897, 948,1147,1192, 2092,1166,1444, 1873)	2970 (3212,2488,3080, 2756,2684,3244, 3076,2568,4072, 2516)	4672 (5440,4548,3940, 3828,3704,6008, 4776,4872,5732, 3876)
NETWORK	execution time	18 (18,18,20,17,16, 17,19,19,19,19)	30 (32,30,31,30,31, 28,31,29,29,30)	45 (46,52,35,48,35, 41,50,55,50,37)
	# of rollback	130 (119,113,119,158, 140,138,127,119, 126,140)	235 (315,242,278,230, 230,117,247,221, 247,221)	505 (557,657,365,575, 391,455,585,558, 531,368)
	# of rolledback transaction	1319 (1359,1254,1438, 1253,1388,887, 1361,1434,1376, 1440)	2459 (3172,2836,2940, 2024,2600,1384, 3036,1956,2176, 2468)	5627 (5532,6724,4352, 6092,4444,5208, 6852,6792,5968, 4304)

Fig.13. The execution of two, three and four Gprocesses

When two Gprocesses were executed simultaneou sly on one processor, the total number of rollbacks was 126 (54+72) and the total amount of simulation backup time was 3386.182 (1117.495+2268.687). The average amount of simulation backup time per rollback was about 26.874 (3386.182 / 126). When two Gprocesses were executed on three processors, the total number of rollbacks was 130

	# of Gpro.		Gprocess1	Gprocess2	Gprocess3	Gprocess4
C	2	rollbacks	54	72		
		transactions	601	750		
		simu. time	1117.495	2268.687		
D	3	rollbacks	78	97	98	
		transactions	903	965	1102	
		simu. time	1126.198	2293.133	8022.754	
S	4	rollbacks	104	113	97	98
		transactions	1026	1130	1280	1236
		simu. time	980.663	2450.206	8874.804	11097.224

	# of Gpro.		Gprocess1	Gprocess2	Gprocess3	Gprocess4
NETWORK	2	rollbacks	63	67		
		transactions	786	733		
		simu. time	1746.955	1883.062		
	3	rollbacks	74	92	69	
		transactions	774	836	849	
		simu. time	1003.048	2359.926	5920.075	
	4	rollbacks	111	144	120	130
		transactions	1341	1432	1448	1406
		simu. time	1595.080	3349.305	9681.135	13267.180

Fig.14. The results of break down

(63+67) and the total amount of simulation backup time was 3630.017 (1746.955+1883.062). The average amount of simulation backup time per rollback was about 26.048 (3630.017 / 130). When three Gprocesses were executed on one processor, the total number of rollbacks was 273 (78+97+98) and the total amount of simulation backup time

was 11442.085 (1126.198+2293.133+8022.754). The average amount of simulation backup time per rollback was 41.912 (11442.085 / 273). When three Gprocesses were executed on four processors, the total number of rollbacks was 235 (74+92+69) and the total amount of simulation backup time was 9283.049 (1003.048+2359.926+5920.075). The average amount of simulation backup time per rollback was 39.502 (9283.049/ 235). When four Gprocesses were executed on one processor, the total number of rollbacks was 412 (104+113+97+98) and the total amount of simulation backup time was 23402.897 (980.663+2450.206+8874.804+11097.224). The average amount of simulation backup time per rollback was 56.803 (23402.897 / 412). When four Gprocesses were executed on five processors, the total number of rollbacks was 505 (111+144+120+130) and the total amount of simulation backup time was 27892.700 (1595.080+3349.305+9681.135+13267.180). The average amount of simulation backup time per rollback was 55.233 (27892.700 / 505). This proves that when the Gprocess increases, the amount of simulation backup time per rollback increases.

In this table, we can see that the simulation backup time of Gprocess2 is larger than Gprocess1's. Also, the simulation backup time of Gprocess3 is larger than Gprocess2's. That is because of the interarrival time of each Gprocess in the GPSS tugboat program. Interarrival time of Gprocess2 is larger than Gprocess1's.

V. Summary and Conclusions

This thesis is concerned with presenting ways of implementing GPSS as a PDES using C-LINDA while solving synchronization conflicts that occur in the implementation process. C-LINDA is a parallel programming language that combines the coordination language LINDA with the programming language C. C-LINDA has several advantages which set it apart from other parallel programming environments. C-LINDA may be used by anyone who has a knowledge of C because C-LINDA augments the C programming language. C-LINDA implements parallelism via tuple space and a small number of operations on it. Tuple space and the operations that act on it are easy to understand and quickly mastered. C-LINDA is available on a large number of parallel computer systems, including shared-memory computers, distributed memory computers, and networks.

The thesis examined the consequences of decomposing a GPSS program into several processes, referred to as workers or Gprocesses, each of which can be run on its own processor. We can determine the number of Gprocesses according to the number of GENERATE statements in the GPSS program. Another control process referred to as a master is required to evaluate each Gprocess, check for synchronization problems, and gather and report the results. The master process maintains a linked list ordered by time of the state of the system each time a statement

that includes a global variable is executed. Every time a Gprocess executes a GPSS block in which a global variable is used a record is added to the linked list in the master process to check if a synchronization problem occurs. To make a rollback possible, every Gprocess maintains a linked list that includes a history of its own transaction movements and a future events list. A tuple, like the local time of the interrupt Gprocess, is stored in the tuple space to use when each Gprocess communicates with the master process for checking the common passive resources and the termination counter if it is used for more than one Gprocess. The master process starts Gprocess which then execute simultaneously.

We have checked each statement of GPSS to see whether the synchronization conflicts can occur when a GPSS program is changed to a parallel program. GENERATE, SEIZE/RELEASE, ENTER/LEAVE, PRIORITY, ADVANCE, STORAGE, START, TRANSFER, DEPART, TEST, INITIAL/SAVEVALUE, ASSIGN, SELECT, TABLE/TABULATE, LOGIC, GATE, and MARK statements were considered. The logic necessary to change a GPSS program to a parallel program was proposed for each of these statements. We have shown how to change a GPSS program to the parallel program using C-LINDA by example.

In this example, we have compared and analyzed the execution time of one processor, three processors(one for the master and two for the Gprocesses), four processors and five processors. And we have compared the execution

time of the termination counters 50, 100 and 500. From all of these cases, we learned several things.

. C-LINDA has a lot of overhead because the execution time was 12 seconds when one processor executed the program without any search for synchronization problems or any rollbacks.

. The search time per 1000 was about 5 seconds regardless of the number of the processors. This was consistent in all executions.

. The backup overhead of about 125 backups and the search time of about 1400 transactions according to rollback was about 5 seconds. This shows that rollbacks require a lot of overhead. That is, that the more rollbacks and transactions rolled back occur, the larger the execution time is.

. When the parallel program was executed with two different processors, the execution time decreased as compared with the execution time on one processor. Also, when the parallel program was executed with several different processors, the decreasing ratio increased as compared with the execution time on two processors.

In conclusion, we broke up the GPSS program into independent Gprocesses. There are cases when it cannot be done, for example, TRANSFER statement from one Gprocess to another.

The results of a parallel program using C-LINDA are as follows :

. C-LINDA has a high overhead on just one processor.

. The more rollbacks and transactions rolled back occur, the larger the execution time is in the parallel program.

.The larger the number of processors are, the smaller the execution time is.

. The larger the number of Gprocesses, the larger the amount of simulation backup time per rollback.

When 2 Gprocesses were executed on one processor and three processors, the amount of simulation time back per rollback was 26 and 26, respectively. When 3 Gprocesses were executed, it was 41 and 39. And when 4 Gprocesses were executed, it was 56 and 55. Using these figures, a limit can be set as to when a process is allowed to proceed. Setting this limit would produce a decrease in rollback and execution time. This also can be the subject of further research.

Bibliography

1. Ahuja, S., Carriero, N. J., Gelernter, D. H. and Krishnaswamy, V., "Matching Language and Hardware for Parallel Computation in the LINDA Machine", IEEE Transactions on Computers, Vol.37, No.8, pp.921-929, 1988.

2. Baezner, D., Cleary, J., Lonow, G. and Unger B., "Algorithmic Optimizations of Simulations on Time Warp", in Distributed Simulation, Vol.21, No.2 (SCS Simulation Series), pp.73-78, 1989.

3. Braunl, T., Parallel Programming : An Introduction, Prentice Hall, 1993.

4. Carriero, N. and Gelernter, D., How to Write Parallel Programs : A First Course, The MIT Press, 1990.

5. Carriero, N. and Gelernter, D., "LINDA in Context", Communications of the ACM, Vol.32, No.4, pp.444-458, April 1989.

6. Cassandras, C. G., Discrete Event Systems : Modeling and Performance Analysis, Richard D. IRWIN, Inc., and Aksen Associates, Inc., 1993.

7. Chandy, K. M. and Misra, J., "Distributed Simulation : A Case study in Design and Verification of Distributed Programs", IEEE Transactions on Software Engineering, Vol.SE-5, No.5, pp.440-452, September 1979.

8. Chandy, K. M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM, Vol.24, No.11, pp.198-206, April 1981.

9. Evans, J. B., Structures of Discrete Event Simulation, John Wiley & Sons, 1988.

10. Fishman, G. S, Principles of Discrete Event Simulation, Wiley-Interscience, John Wiley, New York, 1978.

11. Fishman, G. S. and Moore III L. R., "An Exhaustive

Analysis of Multiplicative Congruential Random Number Generators with Modulus $2^{31} - 1$", SIAM Journal on Scientific and Statistical Computing, Vol.7, No.1, pp.24-45, Jan. 1986.

12. Fujimoto, R. M., "The Virtual Time Machine", Proceedings of the International Symposium (ACM) on Parallel Algorithms and Architectures, pp.199-208, June 1989.

13. Fujimoto, R., "Parallel Discrete Event Simulation (PDES)", Communication of the ACM, Vol.33, No.10, pp.31-53, Oct. 1990.

14. Fujimoto, R. and Nicol, D., "State of the Art in Parallel Simulation", Proceedings of the 1992 Winter Simulation Conference, pp.246-254, 1992.

15. Fujimoto, R. M., "Parallel Discrete Event Simulation : Will the Field Survive?", ORSA Journal on Computing, Vol.5, No.3, pp.213-230, Summer 1993.

16. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V., PVM 3.0 User's Guide And Reference Manual, Engineering Physics and Mathematics Division Mathematical Sciences Section, Feb. 1993.

17. Greenberg, A. G., Lubachevsky B. D. and Mitrani I., "Algorithms for Boundly Parallel Simulations", ACM Transactions on Computer Systems, Vol.9, No.3, pp.201-221, Aug. 1991.

18. Hoover, S. V. and Perry R. F., Simulation : A Problem-Solving Approach, Addison Wesley, 1990.

19. JaJa, J., An Introduction to Parallel Algorithms, Addison Wesley, 1992.

20. Jefferson, D. R., "Virtual Time", ACM Transactions on Programming Languages and Systems", Vol.7, No.3, pp.404-425, July 1985.

21. Jefferson, D. J. and Sowizral H., "Fast Concurrent Simulation Using the Time Warp Mechanism", in Distributed Simulation, Vol.15, No.2 (SCS Simulation Series), pp.63-69,

1985.

22. Kelton, W. D., "Perspectives on Simulation Research and Practice", ORSA Journal on Computing, Vol.6, No.4, pp.318-328, Fall 1994.

23. Kumar, D., "Systems with Low Distributed Simulation Overhead", IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.2, pp.155-165, March 1992.

24. Law, A. M. and Kelton W. D., Simulation Modeling and Analysis, 2nd. Ed., McGraw Hill, 1991.

25. Mehl, H. and Hammes, S., "Shared Variables in Distributed Simulation", in 7th Workshop on Parallel and Distributed Simulation(PADS93), Vol.23, No.1 (SCS Simulation Series), pp.68-75, 1993.

26. Misra, J., "Distributed Discrete-Event Simulation", Computing Surveys, Vol.18, No.1, pp.39-65, March 1986.

27. Nicol, D. M., "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks", SIGPLAN Notices 23, 9, pp.124-137, Sept. 1988.

28. Nicol, D. M., "Conservative Parallel Simulation of Priority Class Queueing Networks", IEEE Transactions on Parallel and Distributed systems, Vol.3, No.3, pp.294-303, May 1992.

29. Nicol, D. M., " Global Synchronization for Optimistic Parallel Discrete Event Simulation", in 7th Workshop on Parallel and Distributed Simulation (PADS93), Vol.23, No.1 (SCS Simulation Series), PP.27-34, July 1993.

30. Nicol, D. and Fujimoto R., "Parallel Simulation Today", ORSA Journal on Computing, pp.1-35, 1994.

31. Nutt, G. J., "Distributed Simulation Design Alternatives", in Distributed Simulation, Vol.22, No.1 (SCS Simulation Series), pp.51-55, 1990.

32. Quinn, M, Parallel Computing : Theory and Practice, McGRAW-HILL, Inc., 1994.

33. Rajaei, H. and Ayani R., "Language Support for Parallel Simulation, in 6th Workshop on Parallel and Distributed Simulation(PADS92), Vol.24, No.3 (Simulation Series), pp.191-192, 1992.

34. Righter, R. and Walrand, J. C., "Distributed Simulation of Discrete Event Systems", Proceedings of the IEEE, Vol.77, No.1, pp.99-113, Jan. 1989.

35. Ronngren, R., Ayani, R., Fujimoto, R. and Das, S., "Efficient Implementation of Event Sets in Time Warp", in 7th Workshop on Parallel and Distributed Simulation (PADS93), Vol.23, No.1, pp.101-108, July 1993.

36. Sahni, S. and Horowitz E., Fundamentals of Data Structures, PITMAN, 1976.

37. Schriber, T. J., Simulation Using GPSS, John Wiley & Sons, 1974.

38. Schriber, T. J., An Introduction to Simulation using GPSS/H, John Wiley & Sons, 1991.

39. Smith, D. S., Brunner, D. T. and Crain, R. C., "Building a Simulator with GPSS/H", Proceedings of the 1992 Winter Simulation Conference, pp.357-360, 1992.

40. Sokol, L. M., Briscoe, D. P. and Wieland A. P., "MTW:A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution", in Distributed Simulation, Vol.19, No.3 (SCS Simulation Series), pp.34-44, 1988.

41. Tomlinson, A. I. and Garg, V. K., "An Algorithm for Minimally Latent Global Virtual Time", in 7th Workshop on Parallel and Distributed Simulation (PADS93), Vol.23, No.1 (SCS Simulation Series), pp.68-75, 1993.

42. Turner, R. S., Patterson, L. I., Hyatt, R. M. and Reilly, K.D., "A Parallel Distributed Simulation System Using Tuple-Space", in 6th Workshop on Parallel and Distributed Simulation (PADS92), Vol.24, No.3 (SCS Simulation Series), pp.193-194, 1992.

43. Wagner, D., Lazowska, E. and Bershad B., "Techniques for Efficient Shared-Memory Parallel Simulation", in

Distributed Simulation, Vol.21, No.2 (SCS Simulation Series), pp.29-37, 1989.

44. West, J. and Mullarney A., "MODSIM : A Language for Distributed Simulation", in Distributed Simulation, Vol.19, No.3 (SCS Simulation Series), pp.155-159, 1988.

45. Xiong, R., "Design of Parallel Mergesort and Quicksort Algorithms", Ph.D Thesis from Graduate School and University Center of City University of New York, pp.5-7, 1990.

46. Original LINDA : C-Linda User's Guide & Reference Manual V2.5.2, Scientific Computing Associates Inc., 1993.