Global States of Distributed Systems: Classification and Applications

Krishnarao Venkatesh

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

January 1988

ISBN 0-315-44876-8

ABSTRACT

Global States of Distributed Systems :
Classification and Applications

Krishnarao Venkatesh, Ph.D,
Concordia University, 1988.

Global information is essential in the operation and application of distributed computer systems. In a distributed system no single process in the system can instantaneously capture the complete system state due to the variable message delay and the autonomous nature of the processes. Global states of different types can be recorded depending on the degree of synchronization enforced between the communicating processes while recording the process and channel states. A formalism for classifying different types of global states of a distributed system is proposed and studied in this thesis. It is based on the characteristics of S-T cuts in the event graph model of distributed computations. Depending on the existence of forward and backward communication edges in these cuts, four main types of global states are identified: Statistical global state, Stable global state, Consistent global state and Synchronized global state. Their properties and applications are studied. New message efficient algorithms are proposed for detection of Consistent, Stable and Statistical global states.

Two applications of distributed systems, namely: Discrete Event Simulation and Backward error recovery, which need global states have been examined in detail. The problem of updating the simulation clock in processes modeled by the strongly connected components of a process graph is viewed as an application of the Global Minimum detection property of Stable global states. Three objectives for efficiently updating the clocks are identified. A new optimal solution is proposed for one of the objectives. The optimization problem for another objective is shown to be NP-Complete. Consequently, a new computationally efficient heuristic solution is proposed. The message complexity and time complexity of these heuristic solutions are shown to be asymptotically better than a known existing algorithm.

Avoidance of Domino effect during Backward error recovery is viewed in the framework of Consistent global state detection. The requirements for avoidance of domino effects are formalized and a new coordinated checkpointing and rollback recovery scheme which eliminates the domino effect has been suggested. In this scheme, simultaneous rollback and recovery can be initiated by a process without it having to wait for the completion of rollback and recovery initiated by other processes.

The theme of the thesis has been to evolve a classification scheme for global states of distributed systems, and examine certain significant applications of these global states.

Dedicated to

Bhagwan Sri Satya Sai Baba

and

Our Parents

ACKNOWLEDGEMENTS

There were significant contributions from a number of people and organizations without which this work could not have been completed successfully. First of all, my sincere thanks to my two advisors Prof. T. Radhakrishnan and Prof. Hon. F. Li. I consider myself singularly lucky to have had the oppurtunity of working with them and sharing their experience. They had to work hard, sometimes even harder than me in order to help me mould my half baked ideas formally and present them precisely. Prof. Radhakrishnan introduced me to Distributed systems and their problems. He was the person who motivated me for doctoral studies for which I am very grateful. I am thankful to Prof. Li who taught me the art of tackling problems in a top down manner and introduced me to formal techniques. The seed for my thesis work was Prof. Li's fascination for blurred pictures: Global states in unreliable communication environments, I mean. Their patience, encouragement and moral support are deeply appreciated.

In this era of differential fees and inflation the financial aspect is the most worrisome issue for international students. I must profusely thank the Canadian Commonwealth Scholarship and Fellowship committee, Govt. of Canada and the Ministry of Education, Govt. of India for awarding me a Commonwealth scholarship and mitigating my financial woes. Miss Deidre Roesar and her excellent staff at the Commonwealth committee took extraordinary care of us and made our stay in Canada a pleasant one. I convey my special thanks to them. The contribution of my employers M/s Bharat Electronics Ltd., Bangalore, India, has been no less important. I am indebted to the management of B.E.L for encouraging me in my doctoral work and sanctioning me leave of absence in order to enable me to complete my studies.

I must thank members and colleagues of the Distributed processing group specifically Heping Shang, Chris Passier and Joachim Hamamjoglu for unhesitatingly devoting the best part of the last six months to developing a Global state detection kernel and a Backward error recovery kernel based on the algorithms presented in this thesis, on a network of Sun workstations. I am sure that with their dedication and perseverance they will be able to complete the implementation work shortly. I am also thankful to Clifford Grossner for giving me valuable insights into the CUENET distributed system, and Thomas Wieland for helping me with an implementation of the global state detection algorithm on CUENET.

As is well known, a picture is worth a thousand words. So I must thank Prof. R. Jayakumar and Dr. A. Selvakumar for helping me convey about 43,000 words of information by lending me their drafting instruments. I thank my fellow students of the Cave, where surprisingly very few discussions are centered on bows and arrows, for providing a stimulating atmosphere for discussions and criticisms. Special mention must be made of Ramachar Prasad, Premchand Nair, Mathew Palakal, Doreswamy Muralidharan and Periyasamy Kasilingam with whom I closely interacted during my tenure here.

How can I express my feelings to the members of my family for their unwavering support, encouragement, faith reposed in me and for prevailing upon me to continue my studies in spite of the calamitous events that happened recently. All I can muster is a simple thank you.

Secretaries of the Computer Science department Ms.Stephanie Roberts, Ms.Halina Monkiewicz, Ms.Terry Czernienko, Ms.Mary Smodis, Ms.Angie De Benedictis and Ms.Irene Mazis cannot be thanked enough for making our interactions with the administration a pleasurable one and also for providing the secretarial support unhesitatingly even when I used to approach them just before the deadlines.

I assure all those persons whom I should have mentioned but have not, that the omission is not intentional and should be excused in view of my memory not being what it was before I started my thesis work.

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ARM	Anticipated Rollback Message list.
BDS	Backward dependent set.
BFT_f	Forward breadth first spanning tree.
BFT_r	Reverse breadth first spanning tree.
CCP	Current Checkpoint vector of a process.
CLA	Chandy and Lamports' global state detection algorithm.
CMU	Carneige Mellon University.
Crc	Create_response checkpoint.
Csic	Create_self_induced_checkpoint.
DCS	Distributed Computer Systems.
DES	Discrete Event Simulation.
DMN	Discard Message Number.
DS	Distribtted System .
DSSX	Descendant Set of a Self induced Checkpoint.
EDS	Error dependent set.
EDSRR	Exclusive Descendant Set of a PRR.
ERL	Effective Recovery Line.
FIFO	First In First Out.
GSK	Global state kernel.
ISDN	Integrated Services Digital Network.

LAN	Local Area Network.
lpo	labelled partial order.
MS	Message state field of a Checkpoint.
NINIT	Number of global state initiators.
PM	Pending Message set.
POMSET	Partially Ordered Multiset.
PRR	Primary Rollback Region.
PS	Process state field of a Checkpoint.
RBR	Rollback Region.
RCP	Checkpoint vector received in a message.
RDS	Retransmission dependent set.
RL	Valid Recovery Line.
RM	Received Message set.
RMN	Receive Message Number.
RPC	Remote Procedure Call.
SC	Set cover.
STM	Space Time Model.
TC	Test message cover.
TOMSET	Totally Ordered Multiset.
VLR	Venkatesh, Li and Radhakrishnans' global detection algorithm.
VM	Valid Message set.
XCP	Checkpoint vector field of a Checkpoint.

$X \xrightarrow{\eta} Y$	State Y is reached from X due to execution of the computation η.
$\preceq$	Simultaneous or precedes.
$\prec$	Strictly precedes.
$\succ$	Succeeds.
$\nexists$	Does not exist.
$\Rightarrow$	Implies.
$\parallel$	Concurrent.

Chapter 1

Introduction

Practical distributed computer systems (DCS) have evolved due to the recent advances in semiconductor and communication technologies. Wide availability of experimental distributed systems has stimulated interest in their applications. In a DCS numerous independent and asynchronously operating process-memory pairs are interconnected by means of high bandwidth communication channels. Each process-memory pair has a local clock and there is no global clock controlling the whole system. Interprocessor communication is achieved by passing messages and not via shared memories. The propagation of messages takes a finite but variable amount of time. In a typical DCS, this message propagation time is usually very much greater than the individual processor cycle time. As a consequence of this propagation delay there exists a delay δt between the time at which a fact is known in one processor and the time at which the complete network becomes aware of this fact. This peculiarity of distributed systems has significant influence on the nature of the system wide information that can be practically gathered, as due to this delay instantaneous system wide pictures cannot be captured.

A wide spectrum of applications exist for distributed systems as they make a large number of processor cycles available by using a number of powerful processing elements, and also allow geographical distribution of these cooperatively functioning intelligent resources to the locations where they are mostly needed. Discrete Event Simulation which has a high proportion of concurrent activities is an ideal candidate for exploiting the processing power of a DCS and hence has evolved as one of its important applications. The possibility of geographical distribution of processors and the potential of higher system availability make distributed systems ideal for control of critical and often life threatening applications like chemical process control, Aircraft control etc.

The peculiar properties of distributed systems and their practical utilization have posed a number of interesting basic problems and have spurred active research on fundamental topics in this area. This work focusses on the basic issue of global states and considers practical ramifications in the areas of Discrete Event Simulation and, Rollback and Recovery of distributed systems.

1.1 Focus of the thesis

Distributed systems in general have no single master site for global control. However, efficient solutions for a number of problems in distributed computing require gathering of information from the whole system. The lack of a common clock, the variable propagation delay of the communication subnetwork and the asynchronous nature of the processes make it impossible for any single observer to capture the instantaneous state of the whole system. Typically a global state is composed of local states captured at the processes along with information contained in messages in transit. The exact content of the process states and the required interrelationship between these captured states of processes is application dependent and results in the existence of different types of global states. In this context certain important issues have to be addressed.

(a) If all global states are not the same then how could they be classified?

(b) Are there efficient algorithms for determining these global states?

These issues have not been addressed satisfactorily till now. Recent interest in them has been triggered by the seminal work on formalization of the concept of global states done by Chandy and Lamport [11]. However, the problems have not been dealt in their entirety.

Application specific aspects of global states and the relevant issues of management of global information are of particular significance in important applications like Discrete Event Simulation and Fault tolerant distributed systems. In Discrete Event Simulation (DES) the simulation model is divided into interacting submodels which are assigned to individual processes. Each process has its own logical clock. The basic activity pursued by the process is simulation of events. Frequently the simulation models contain strongly connected sets of processes. In this context the main issues to be addressed are:

(c) How should the logical clocks of the processes be updated such that the correctness of the simulation is preserved?

(d) Are there optimal ways of doing this?

The issue of updating logical clocks during discrete event simulation of models with Strongly connected components, in the graph theoretic sense, has been addressed by Bryant [8]. However, the treatment is ad hoc and the algorithm is inefficient. These interesting questions are methodically analysed here and solutions are proposed.

Design of fault tolerant distributed systems must address the issue of failures caused by unanticipated faults like design flaws which occur fairly frequently. In case of failures, the dynamics of the process under control and the extent of. computation already completed often require the computation to be resumed with minimal loss. Usually errors of this nature are handled by rolling back the system to a known non-faulty state and commencing re-execution from this state. This requires saving the state of the system at different points during the computation. In this context the important issues to be addressed are:

(e) When and how should the system state be saved?

(f) How should the rollback be performed efficiently?

These issues on Rollback recovery have been extensively researched in the past decade. One of the main research findings is that unless sufficient care is taken during the state saving operation, a phenomena called "domino rollback" could occur when an attempt is made to rollback the system to a state in which all effects of errors are overcome and from where meaningful computation can proceed. Domino rollback typically undoes a lot of correct computations and sometimes might rollback the system to the beginning. Numerous solutions have been proposed for domino effect free rollback. Some of them restrict the allowable interprocess communication patterns, while some others create useless checkpoints or unnecessarily save all interprocess messages.

In this thesis the above mentioned three seemingly independent problem clauses have been unified based on their requirement for global information. The solution to the problem of updating clocks in discrete event simulation, requires the assessment of the largest increment possible in the whole system with the constraint that simulation of none of the events in the system is missed. This requires capturing the set of all events currently in the system either in processes or in transit, in order to determine the event with the earliest simulation time. The global state recording and compilation phases can be effectively combined to evolve efficient schemes for clock update.

Interrelated checkpoints will have to be established in the interacting processes in order to avoid domino effect during rollback and confine the rollback to the affected processes. This, once again, requires establishment of global information in the process of checkpointing the original computation, and retrieval of this information during rollback.

Thus the main ideas in the thesis are about efficiently capturing the required global information and effective utilization of this global information in the various applications of interest.

1.2 Contributions

This thesis answers the questions raised in the previous section. In this work major effort was concentrated on:

(a) Evolution of a classification scheme for global states.

(b) Efficient handling of application specific requirements for management of global information in two important applications namely Discrete Event Simulation and Backward Error Recovery.

The key contributions of this thesis are briefly outlined below.

Applications making use of global information were analyzed to determine the nature of the global information used. Based on this analysis a classification scheme has been evolved. Four different types of global information or states have been identified namely: Statistical global states, Stable global states, Consistent global states and Synchronized global states. Significant properties of these global states were examined after formulating generic problems for each category.

Efficient algorithms which minimize number of control messages were developed for detecting these global states. These algorithms were designed to be applicable without the FIFO restriction on the channels employed by previous algorithms, which significantly decreases the complexity of the communication subsystem and improves the throughput. The message complexity of our algorithm is O(n) while that of comparable existing algorithms is O(L), where n is the number of processes and L is the number of channels.

The issue of updating local clocks in a strongly connected subset of processes, while performing discrete event simulation is viewed as an extension of the generic problem corresponding to Stable global states. A simple solution to this problem exists based on the observation that certain crucial information (in this case the simulation time of events) can be preserved at the sender's site instead of discarding this information at the sender's site and trying to recreate the same information at the receiver's site later on. The problem of determining the earliest event in the strongly connected component with minimal message overhead is shown to be NP- complete. Hence heuristic solutions have been presented for this problem. The message complexity of these heuristic solutions is O(n) while that of existing algorithms is $O(n^3)$.

The problem of rollback recovery is viewed as an extension of the generic problem corresponding to Consistent global states. A checkpointing and rollback recovery scheme to achieve domino free rollback has been suggested. This scheme minimizes rollback and restricts the rollback to only the affected processes by tracking dependencies between processes dynamically. Domino effect is avoided by coordinated checkpointing. The scheme has been designed to support concurrent rollbacks. Rollbacks occur without requiring the system to be frozen during rollback. This scheme achieves minimal rollback, does not restrict communication patterns, supports nondeterministic computation, also saves just the messages required for playback during re-executions and thus shows a significant improvement over existing schemes.

1.3 Organization

The thesis is organized as follows. The basic distributed computer system model used, and the models employed for representing distributed computations are presented in Chapter 2. A global state classification scheme along with generic problems and their relevant properties are presented in Chapter 3. Efficient algorithms for detecting global states and their performance characteristics are discussed in Chapter 4. The problem of clock update in discrete event simulation of strongly connected models is the topic of Chapter 5. Heuristic solutions and their performance are presented in this chapter. The problem of rollback recovery is the subject of Chapter 6. The reasons for domino effect in rollback is formalized. A dependency based checkpointing scheme and a rollback recovery algorithm are presented. Also the applicability of this algorithm to concurrent rollbacks is demonstrated in this chapter. Conclusions of the thesis and certain suggestions for future work are presented in Chapter 7. Finally some examples helpful in the illustration of the concepts in the text and proofs of some theorems on global states constitute the two appendices of the thesis.

Chapter 2

Models of Distributed Computations

Models facilitate analysis and verification of systems by providing tools for precise definition of specific properties and characteristics of these systems. In this chapter two formal models of distributed computations namely: the Space Time Model and the POMSET model used in this thesis are presented. The graphical nature of the Space Time Model and the mathematical richness of the POMSET model complement each other to make presentation and formalization of ideas simple.

2.1 Distributed System model

A distributed system consists of a finite number of interconnected communicating sequential processes. The following initial assumptions are made regarding their operational characteristics.

Assumptions about Channels

(CA1) Interprocess communication is through point to point directed communication channels.

(CA2) Channels are reliable and ensure FIFO delivery,

(CA3) The delay incurred in sending a message through a channel is finite and variable but positive (causality).

(CA4) Unbounded numbers of message buffers exist in each channel.

Assumptions about Processes:

(PA1) Processes communicate via messages through explicit send and receive events.

(PA2) An event e of process P is denoted by a quintuple $\langle P,s,s',M,C \rangle$. Such an event is atomic and changes the state of process P from s to s′ upon receiving or sending message M on channel C, depending on whether C is an input or an output channel respectively. In case the M and C in e are empty, e represents an internal event of process P. Every event is completely executed in a single process.

(PA3) No global clock is present.

(PA4) No assumption is made on the order in which the process accepts messages from different channels. However, within each channel messages are delivered in a FIFO manner,

Linguistic support

The programming language supporting distributed programming is assumed to support nonblocked send and receive functions. In addition a message test primitive useful for determining the presence of messages on channels is also assumed to be present. Blocked receives can be realized by combining the message test primitive with the Receive primitive.

2.2 Model of the distributed computation

Both the Space time model (STM) [1], and the POMSET [36] model are used to represent distributed computations in this thesis. The graphically oriented STM model is used to express the ideas and concepts simply, while the mathematically rich POMSET model is used to formalize these concepts. Both models express computations as relations between partially ordered events. A brief review of the aspects of the models relevant to this work follows.

2.2.1 Space Time Model

The system is viewed as a collection of processes. Each process consists of a sequence of events. Each event gets executed in a single process. The events are labelled by the actions they perform. This labelling depends on the application and an event could represent execution of a function or the execution of a single instruction. Sending and receiving of messages are assumed to be events in the processes and interaction between processes is assumed to be only through messages. A relation $\prec$ is defined such that :

(a) If a and b are events in the same process then $a \prec b$ implies that a precedes b.

(b) If a is a send event and b is the corresponding receive event then $a \prec b$ due to causality.

(c) If $a \prec b$ and $b \prec c$ then $a \prec c$ - transitivity.

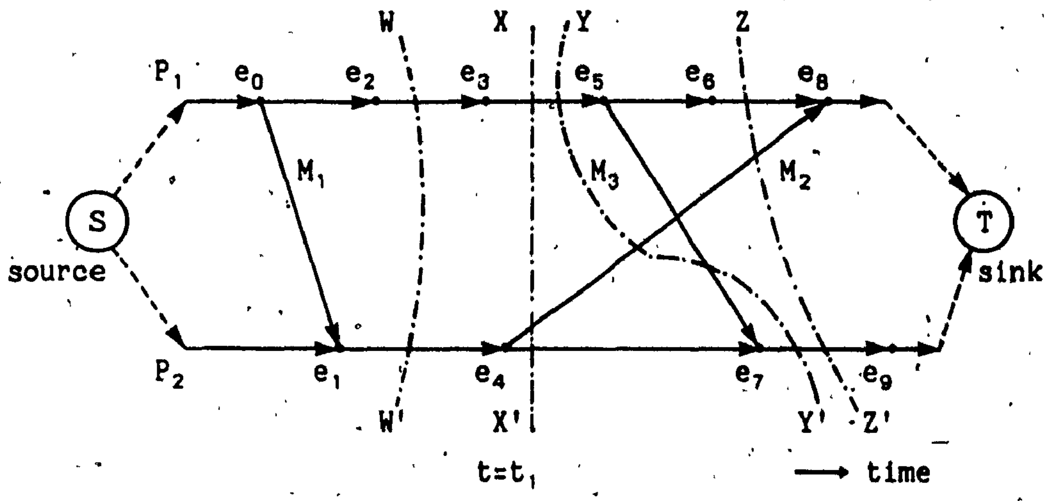
(d) Two distinct events a,b are concurrent if neither $a \prec b$ nor $b \prec a$.

In short, the model assumes that

(i) All events in a process are totally ordered.

and (ii) Interprocess temporal orderings are governed by the necessity of the send events to precede the corresponding receive events.

This can be viewed in terms of a two dimensional space time diagram (event graph) as shown in fig.2.1. The space is represented in the vertical direction (processes P_1, P_2) and the time in the horizontal direction. The circles (nodes) represent events, and all events that occur in a process are in the same horizontal line representing that particular process. For two events a and b in a process P_i, node corresponding to event a occurs before (w.r.t time) event b iff $a \prec b$. Transmission of messages are represented by edges linking send events with the corresponding receive events. By introducing two special nodes (S and T) and connecting them to the horizontal process lines as shown in fig.2.1, the event graph is transformed into a two terminal S-T event graph. An S-T cut of the graph is formed by a set of edges whose removal disconnects node T from S. The cut partitions the event graph into two distinct parts. The part that includes the S-node is known as the S-partition and the part that includes the T-node is known as the T-partition. The terms S-T graph and event graph are used



M_2 : Forward edge w.r.t Cut XX',Cut YY',Cut ZZ'.

M_3 : Backward edge w.r.t Cut YY'.

Cut XX': Instantaneous cut at t= t_1

Cut YY': Stable cut

Cut ZZ': Consistent cut

Cut WW': Synchronized cut

Figure 2.1: Event graph

interchangeably to refer to an S-T event graph. For this model the following are defined.

Definitions:

- Each edge along a horizontal (process) line in the event graph is a *Process edge.*
- A cross edge between events on two different horizontal lines is a *Channel edge.*
- A channel edge intersecting a cut is a *forward edge* if it emanates from an event node in the S-partition of the cut and terminates in the T-partition, else it is a *backward edge.*
- A cut is an *Instantaneous cut* if it is a vertical cut obtained at a particular time instant, say t_i in the event graph.
- A cut is a *Consistent cut* if it does not contain backward edges and intersects precisely one edge on each process line.
- A cut is a *Stable cut* if it intersects precisely one edge on each process line.
- A Stable cut is a *Synchronized cut* if it does not contain any channel edges.

In fig.2.1, Cut XX′ is an Instantaneous cut obtained at time t_1. Cut YY′ is a Stable cut and is not a Consistent cut as it includes a backward channel edge (e_5,e_7). Cut ZZ′ is a Consistent cut. Notice that an Instantaneous cut is always Consistent, because of causality in message communication. Cut WW′ in fig.2.1 is a Synchronized cut.

2.2.2 POMSET model

Concurrency can be expressed by making use of the partially ordered multiset model proposed by Pratt [36]. It is formally defined as follows

- A *partial order* $(V,\preceq)$ is an irreflexive transitive binary relation on a vertex set V, where V is the set of events.
- A *labelled partial order* (lpo) is a quadruple $(V,\sum,\preceq,\mu)$ where $(V,\preceq)$ is the partial order, $\sum$ the set of actions and μ: $\sum \rightarrow V$ labels the vertices of V with the symbols from the alphabet $\sum$.
- A *pomset* $[V,\sum,\preceq,\mu]$ is defined as the isomorphism class of the lpo$(V,\sum,\preceq,\mu)$. Two lpos $(V,\sum,\preceq,\mu)$ and $(V',\sum,\preceq',\mu')$ are said to be isomorphic provided there exists a bijection τ:V $\rightarrow$ V′ such that $\forall$ u $\in$ V, $\mu(u) = \mu'(\tau(u))$ and $\tau(u) \preceq' \tau(v)$ only when $u \preceq v$. Note here that the importance is given to preserving the temporal ordering of the actions, and the identities of the vertices are themselves unimportant.

A set of Pomsets characterize a process. This set will represent all possible behaviors of the process. An algebra of operations have been defined by Pratt for Pomsets. Complete treatment of this can be found in [36], only some of the operations on Pomsets relevant to our study are defined below:

<u>Concurrence</u>: The concurrence of two pomsets $p:[V,\sum,\preceq,\mu]$ and $p':[V',\sum',\preceq',\mu']$ is the pomset $[V\cup V',\sum\cup\sum',\preceq\cup\preceq',\mu\cup\mu']$. There is

no interference between the two and hence no violation of transitivity.

<u>Concatenation</u>: The concatenation $p;p'$ of the two pomsets p and p' is the same as concurrence except that the partial order will be $(\preceq \cup \preceq' \cup V \times V')$. All events of p precede events of p'.

<u>Prefix</u>: A pomset p is the prefix of another pomset q provided p is realized from q by deleting events from it such that if an event is deleted then all its successors are also deleted. Henceforth, in this thesis this type of prefix will be referred to as a Consistent prefix.

<u>Temporal operator</u>: A temporal operator ρ is defined which translates a first order predicate ϕ to a predicate ψ such that $\psi(u)$ is true provided $\exists v$ for which $\phi(v)$ is true and $u \preceq v$. i.e. there exists at least one successor v of u for which the predicate ϕ holds. This is represented as $\rho\phi$. Additionally, another temporal operator γ is defined which is similar to ρ but in this case the predicate ϕ holds for all future events. ρ^{+} implies that ϕ holds good if strict future is considered, i.e. $\psi(u)$ is true provided $\exists v$ such that $\phi(v)$ is true and $u \prec v$. $\rho^{\pm}$ indicates temporal reversal.

Network of Processes

The individual processes can be interconnected to realize a system of communicating processes. The behavior of the composite process is expressible as a pomset and the restriction of this behavior to the individual processes must result in valid pomsets for each of the constituent processes. In the application space of interest, a network of communicating sequential processes is assumed in which the events are all atomic and hence the behavior of each constituent process can be individually expressed as a totally ordered multiset (tomset). However, the behavior of the composite system will in general involve concurrent events in the constituent processes and hence would have to be expressed as pomsets. In this context the following can be defined.

Colocation: If events are tagged with a location id, such as a port of a process or a process of a network, then all events with the same tags are said to be colocated.

Some new operators which are needed for our study are introduced.

Stable prefix: A Stable prefix p of a pomset q is one in which some elements of q have been deleted such that if an element is deleted from q then all its colocated successors are also deleted from it.

Synchronized prefix: A Synchronized prefix p of a pomset q is a Consistent prefix in which the elements of q have been deleted such that if a send or receive event e is deleted from q then the matching receive or send event m(e) is also deleted from it.

Instantaneous prefix: If a global clock were available the Instantaneous prefix p at t_1 of a pomset q is derived from q by deleting only those events in q which were executed after t_1. This can be formally expressed as follows, where $\phi_{t_1}(u)$ is defined to be true if event u is executed after t_1:

$$p \preceq_{ip} q = (p \preceq_{\pi} q) \wedge (\neg\rho^{\pm} \phi_{t_1}) \wedge (\gamma\phi_{t_1}).$$

The first term on the right hand side states that p is a consistent prefix, the second term states that for no event u included in p the predicate holds good and the third term states that for all events of q not in p the predicate holds good.

2.2.3 Relationship between the two models

The STM and POMSET models are essentially equivalent (have similar expressibility) in the restricted case of a network of communicating sequential processes. A cut in the STM model is characterized by the computation that has elapsed from the beginning of the computation until the cut. We refer to this as *characterization computation* of the cut. The *state of a cut* is the state of the processes and the channels, acquired after execution of the characterization computation of the cut. These characterization computations can be expressed in terms of pomsets. In this context the following can be trivially inferred:

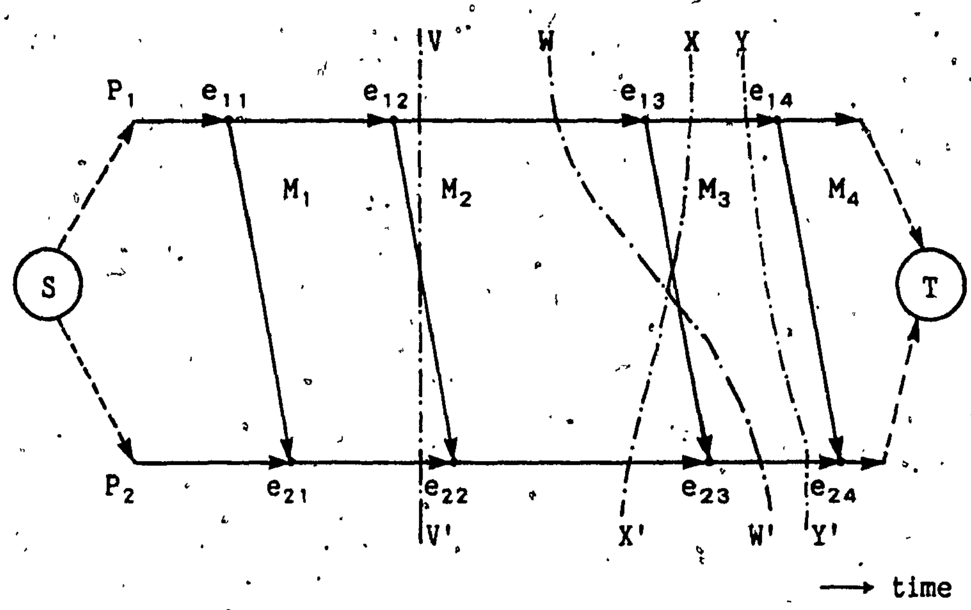
(1) Characterization computation of a Stable cut is a Stable prefix.

(2) Characterization computation of a Synchronized cut is a Synchronized prefix

(3) Characterization computation of a Consistent cut is a Consistent prefix

and (4) Characterization computation of an Instantaneous cut is an Instantaneous prefix

These relationships are invoked in the discussions on global states and the rollback recovery scheme. These correspondences are illustrated in fig.2.2.



A Pomset representing this computation

:$e_{11}(e_{21} \| e_{12})(e_{22} \| e_{13})(e_{14} \| e_{23})e_{24}$

VV' represents an Instantaneous Cut corresponding prefix

$= e_{11}(e_{21} \| e_{12})$

WW' : Stable Cut, Corresponding Stable prefix

$= e_{11}(e_{21} \| e_{12})e_{22}e_{23}$

XX' : Consistent Cut, Corresponding Consistent prefix

$= e_{11}(e_{21} \| e_{12})(e_{22} \| e_{13})$

YY' : Synchronized Cut, Corresponding Synchronized prefix

$= e_{11}(e_{21} \| e_{12})(e_{22} \| e_{13})e_{23}$

Figure 2.2: Correspondence between STM and POMSET models

Chapter 3

Global State

Conceptually, the state acquired by the distributed system after execution of each event of a distributed computation can be interpreted as an instantaneous global state provided the events of the computation are totally ordered. In this case the distributed computation itself can be expressed as a sequence of events. It will then constitute a Totally Ordered Multiset or TOMSET.

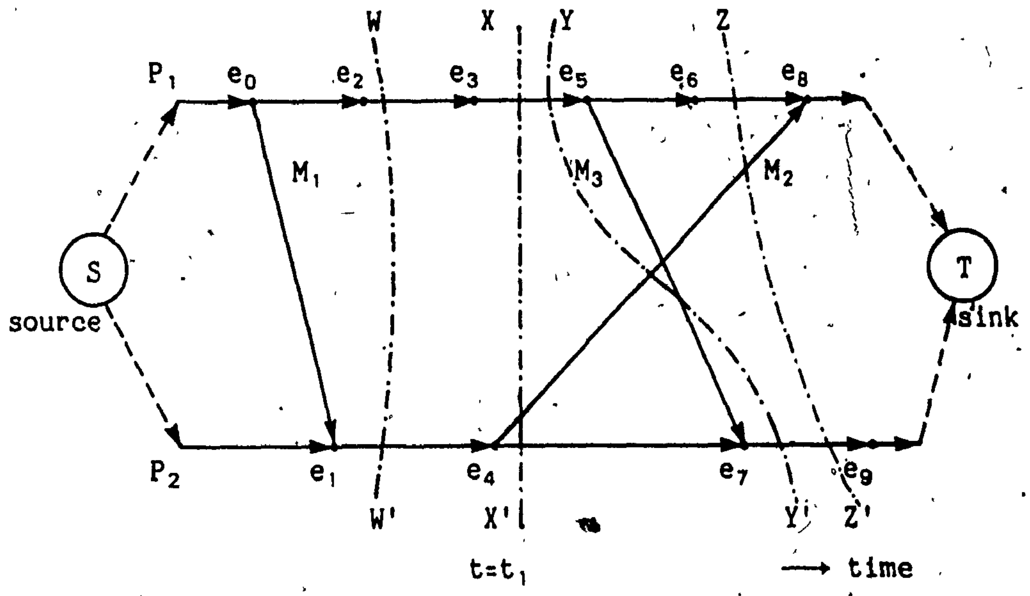
An instantaneous global state S of a distributed system is composed of the local states of all its constituent processes and the states of all channels. The initial global state S_0 specifies that all the constituent processes (P_i) are in their initial states s_o^i and all the channels are empty. The global state is altered by the occurrence of events in the constituent processes. A read event e_j: $\langle P_i, s_k^i, s_n^i, M, C_l \rangle$ can occur in the global state S if the state of P_i is s_k^i and M is at the head of the buffer of input channel C_l of P_i. After the occurrence of this event the global state of the system will change to S′ which is different from S and is obtained by changing the state of P_i to s_n^i and excluding M from the state of C_l. Similarly, a write event can be envisaged. Let a state transition function "next" specify the change of the global state upon the occurrence of an event. For the above example, $S' = next(S,e_j)$. A sequence of events Seq: $(e_0 e_1 \ldots e_r)$ is said to be a computation of the system if the resulting global state:

$$S_{i+1} = next(S_i,e_i) \text{ for all } i \text{ in the range } 0 \leq i \leq r.$$

The sequence Seq represents a serial schedule of a possible execution on the system obtained by viewing the event graph as a precedence graph (fig.3.1).

A number of applications and control problems in distributed systems require the gathering of global system information for taking decisions or engaging in further computation. However, as instantaneous global state cannot be realistically obtained, a suitable global state must be pieced together using recordings of local states of processes and channels, depending on the application requirements. In order to obtain such global information, typically one or more of the processes are designated as *coordinators*. The coordinators acquire global information and compile it for decision making. A two-phase processing results. In the first phase, called the *probe phase*, the distributed processes are coordinated to perform a local state recording. Because of the distributed characteristics of the system, such recordings are not performed simultaneously. In the second phase, called the *compilation phase*, the coordinator gathers all local recordings and composes them to form a global state.

The semantics of the global state is application dependent. Well-known applications which have attempted to make use of the global state concept are listed in Table 3.1. Based on the problem requirements and the tightness of coordination in local state recording, the global state



M_2 : Forward edge w.r.t Cut XX',Cut YY',Cut ZZ'.

M_3 : Backward edge w.r.t Cut YY'.

Cut XX': Instantaneous cut at $t= t_1$

Cut YY': Stable cut

Cut ZZ': Consistent cut

Cut WW': Synchronized cut

Seq: $e_0e_1e_2e_3e_4e_5e_6e_7e_8e_9$

Figure 3.1: Event graph

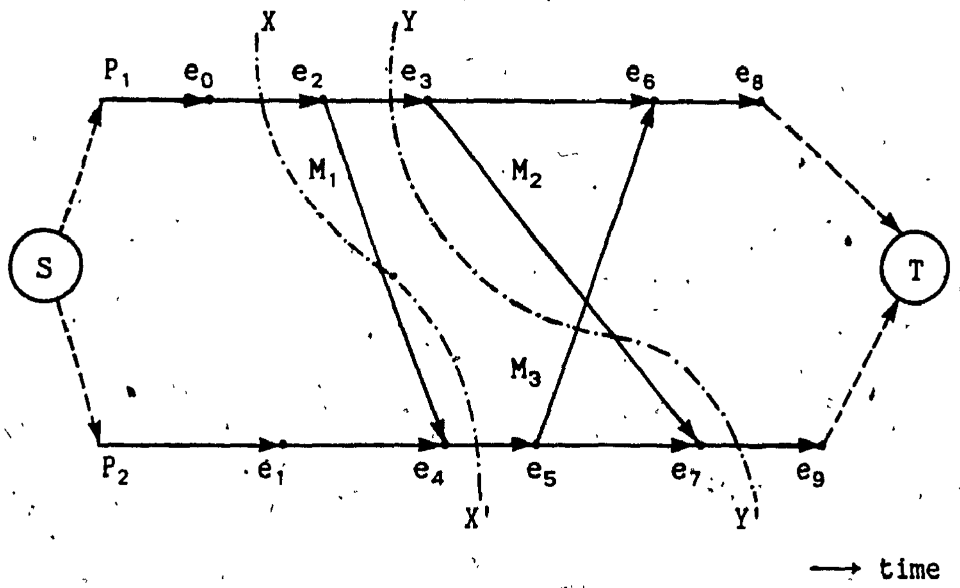
Group	Application	Global Information required
(a)	1. Task Scheduling [45] 2. Load Balancing [53] 3. Query Allocation [9] 4. Performance monitoring 5. Network Status Maintenance [12]	CPU load, I/O load Mean Response times, Mean queue lengths etc.
(b)	1. Termination detection of distributed computation [16] 2. Deadlock detection [17] 3. Distributed garbage collection [31] 4. Discarding Obsolete information in DDB [40] 5. Updating Simulation Clocks in DES [49]	State of the individual processes and knowledge of the type / content of messages in transit.
(c)	1. Rollback Error Recovery [25] 2. Dynamic Resource Allocation 3. Distributed debugging [20,47]	Same information as in (b) but requires tighter coordination than (b)
(d)	1. Synchronous Checkpointing and Rollback Error Recovery [38] 2. Checkpointing distributed databases [18, 42]	State of individual processes recorded with all processes mutually synchronized.

Table.3.1.: Well known distributed system control problems

can be classified into four main groups: (a) Statistical global state, (b) Stable global state, (c) Consistent global state and (d) Synchronized global state.

3.1 Statistical Global State

In this class of applications, the processes are probed in order to capture process status information such as CPU load and I/O load. Here the processes are often the distributed kernel of the operating system. Decisions such as allocation of tasks and queries are based on statistical information like the job queue length, estimated available CPU processing time, message delays etc. The required statistical information can be captured by a probe phase which performs the recording of all process states within a close proximity in time corresponding to the chosen cut. The information on the process edges of the cut are recorded while that on channel edges are ignored as it is assumed that the messages in transit will not significantly affect the statistical values being recorded. Such a Statistical global state is exemplified in fig.3.2 by the state of cut XX′. Only process edges intersected by the cut are of significance here even though some cross edges might also be intersected. Such a cut which contains only process edges is known as a *process cut*.



Statistical Global State : State of cut XX'
= <State of P_1 after e_0,
State of P_2 after e_4 >

Stable Global State : State of cut YY'
= <State of P_1 after e_2, State of P_2 after e_7,
$Channel_{12}$ = NULL, $Channel_{21}$ = M_3 >

Figure 3.2: Statistical and Stable Global States

3.2 Stable Global State

This class of applications center around *stable properties* [11]. To reveal the implicit requirements of stable properties, the following formalism is introduced.

Definition:

- A process is in a *Stable (Local) State* if it will not execute any further Send-event unless it receives some specific message (successful receive-event).
- A distributed system is in a *Stable global state* if:

  (St1): all constituent processes are in stable state (local),

  and (St2): none of the messages in transit on some channel will lead any process to a non-stable state.

The above definition has an important implication.

Lemma 3.1: A distributed system will continue to be in a Stable global state once it has entered one.

Proof: Immediate from the definition, as each process continues to be in a stable state and eventually all channels are flushed.

Q.E.D.

The above notion of Stable global state is useful in problems such as termination detection, deadlock detection and garbage collection. It can actually be strengthened as explained in §3.2.2, to solve similar problems such as the removal of obsolete information in replicated databases [40], and update of simulation clocks in distributed discrete event simulation.

Definition

A *Stable cut recording* is the recording of local process state and channel state corresponding to information conveyed in a stable cut of the S-T graph. The exact attributes contained in these state recordings are application dependent. A Stable cut recording is exemplified in fig.3.2 by the state of cut YY′.

Theorem 3.1

If a distributed system (DS) is in a Stable global state then a Stable cut recording will reveal it.

Proof: Suppose at time t_s, DS is in a Stable global state (i.e., the instantaneous cut at time t_s corresponds to a Stable global state) and a Stable cut recording commences at time $t_r \geq t_s$. From lemma 3.1, we know that the stable cut recording will include process states which must be (locally) stable (St1). Therefore, this Stable cut recording fails to reveal the global stable state property only if some message contained in a channel state recording will lead some process to a future non-stable state. But the same message must have been sent before t_s and should also appear in the instantaneous cut at time t_s. The latter condition forces DS to be in a globally non-stable condition at time t_s, which is a contradiction. Thus a stable cut after t_s should reveal the global stability.

Q.E.D.

Besides the ability to reveal the existence of global stability, a Stable cut recording should not erroneously reveal one when it is not there. To prove this result, we first establish the following lemma.

Lemma 3.2: After participating in a Stable cut recording which satisfies both (St1) and (St2), a process will not send any more messages on a channel.

Proof: This follows from the fact that a process in stable state can enter a non-stable one (so as to send messages) only if it has received a message to enable it to make the state transition. Condition (St2) ensures that no process will ever receive such a message.

Q.E.D.

Theorem 3.2.

If the Stable cut recording completed at time t_f contains process states and channel states that satisfy (St1) and (St2) respectively, then the system DS is globally stable (instantaneously) at time t_f.

Proof: If the instantaneous global state at time t_f is not globally stable and violates either (St1) or (St2), it could only be caused by a message sent by some process after the Stable cut recording. But this violates lemma 3.2.

Q.E.D.

3.2.1 Applications of Stable global state:

It is rather obvious to see how Stable cuts can be applied to solve the termination detection, garbage collection, and deadlock detection problems. In particular, in the case of termination detection the process states are precisely the "active" and the "sleep" states of the process, while the channel state will correspond to the number of messages in transit on the channels. If in a Stable state recording all processes are in the sleep state and there are no messages in transit, then the termination condition is satisfied. In the case of garbage collection the process states will be the "currently referenced objects" in the processes, and the channel states would be the "transfer of reference" messages in transit. So an object can be detected to be [illegible]less, if in a Stable global state recording none of the processes have a reference to this object, and there are no messages with a reference to this object in transit. In the case of deadlock detection the process state is the "running" and the "blocked" states of the processes, and the channel states are the "Resource grant" messages in transit. So if in a recorded state a cyclic dependency of blocked processes is detected with no grant messages coming into any of these blocked processes then a deadlock can be inferred. In summary, the system is globally stable iff the processes are locally stable and the channels are either empty or do not contain the information needed to change the state of any process.

3.2.2 Extension of Stable Global State to determine Global Minimum

In a number of applications, detection of global minimum among time-varying distributed objects is involved. We will present an abstract formulation of the problem and demonstrate how a stable cut recording can be used to solve this problem.

Consider the distributed system (DS) which maintains a set of distributed objects each with a unique label (for example time-stamp). A process consumes an object with the smallest label (r_i) and consequently may reproduce one or more objects with labels $r_o \geq r_i$ to be sent to other processes or added to its own list of objects. At some point in time, a coordinator process conducts a probe phase to decide the minimum label of all objects in the distributed system. This abstract problem is applicable to discrete event simulation [49] where, the objects are events yet to be simulated, and the global minimum corresponds to the maximum clock advancement allowed in the simulation system. Similar adaptation of this abstract problem can be derived to solve the problem of discarding the obsolete information in a replicated distributed database [40].

The stable cut recording can be extended as explained below to solve the global minimum detection problem. Let us assume that the global minimum is decided (compiled) by examining all objects either present in a local process (recorded in the process state) or present in a channel (recorded in the channel state).

Theorem 3.3

The global minimum g_p obtained by recording a stable cut satisfies the relationship: $g_s \leq g_p \leq g_f$.

where $g_s =$ Global minimum associated with the instantaneous cut at the time t_s when stable cut recording commences.

$g_f =$ Global minimum associated with the instantaneous cut at the time t_f when stable cut recording completes.

Proof:

Case(1): Suppose $g_p < g_s$.

If g_p is caused by an object that exists at time t_s then $g_s = g_p$ thus resulting in a contradiction. On the other hand, if g_p is caused by an object produced in the time interval (t_s,t_f), then its label must be $> g_s$. ("causality" of label). Thus $g_p \geq g_s$.

Case(2): Suppose $g_p > g_f$.

This again is a contradiction as an object used in the determination of g_p either is used also to determine g_f or is already consumed and its label is strictly not greater than those yet to be consumed. Thus $g_p \leq g_f$.

Q.E.D.

In summary, a stable cut can be used to detect stable global properties as well as the global minimum property in distributed systems.

3.3 Consistent Global State

Following the notation introduced earlier, we could characterize a recorded global state (S^*) corresponding to some cut as Consistent according to the reachability of that state from the initial state (S_0).

Definition:

A recorded global state S^* is Consistent if there exists a sequence of events $Seq = (e'_1, e'_2, \ldots e'_k)$ so that

$$S^* = next(next \ldots next(next(S_0, (e'_1, e'_2, \ldots e'_k) \ldots).$$

Before exploring the useful properties of a Consistent global state, we first establish its relationship with a Consistent cut.

Theorem 3.4

The recorded global state S^* corresponding to a Consistent cut must constitute a Consistent global state.

Proof:

Part 1: The state associated with a cut which contains a backward edge is not a Consistent global state.

Consider the event graph exemplified in fig.3.3 which represents a computation that has occurred during execution. Let XX′ be a cut of the graph which contains a backward channel edge (e_c, e_d) w.r.t. XX′.

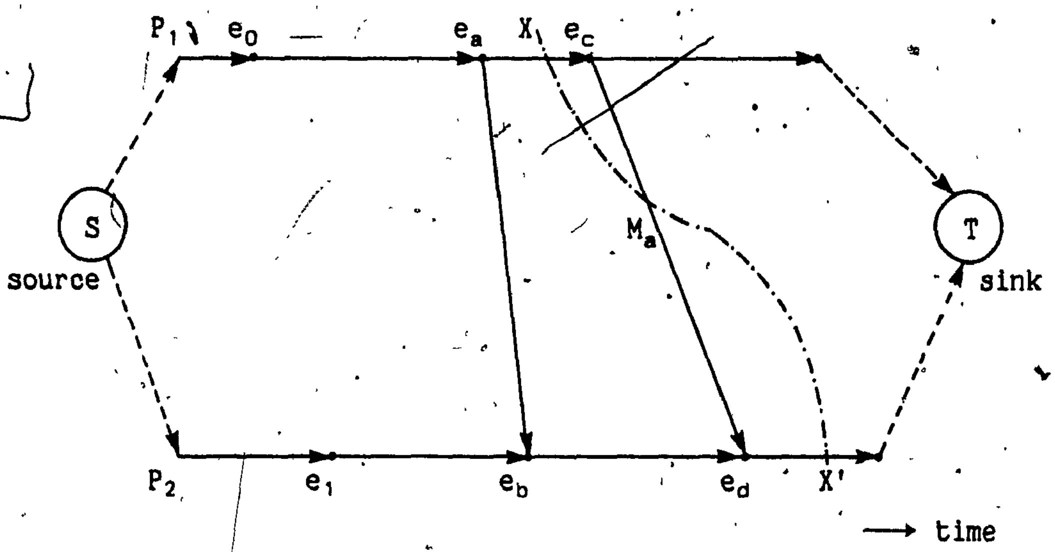


Figure 3.3: A S-T cut containing a backward edge

Assume that the state of XX′ recorded is a Consistent global state.

The backward channel edge (e_c, e_d) signifies that e_c is a predecessor event of e_d. More specifically, e_c is a send event and e_d is a receive event. If the recorded state associated with the cut is a Consistent global state, there must exist a sequence of events in the processes of the form Seq: $(e_1', e_2', \ldots e_k')$ leading the system into the recorded state (definition). But since $e_c \notin$ Seq, which implies that $e_d \notin$ Seq, else causality is violated. This is a contradiction.

Part 2: The state associated with a cut containing only forward edges (from S to T partitions) must be a Consistent global state.

Since the S-partition contains an acyclic subgraph of events, we could trivially traverse these events in a list schedule and such a schedule forms a feasible computation leading the system to the recorded state.

Q.E.D.

Corollary 3.1: The state of the system (processes and channels) reached after executing the set of events which constitute a Consistent prefix of some valid computation is equivalent to a consistent global state associated with the corresponding consistent cut.

Proof: This follows directly from the definition of a consistent global state and observing that after the execution of the prefix, the channels will contain messages which have been generated by send events in the prefix but whose matching receive events are not in the prefix. This corresponds to forward cross edges intersecting the consistent cut. The states attained by the processes is equivalent to those recorded for the corresponding consistent cut.

Q.E.D.

3.3.1 Properties of the recorded Consistent global state

As explained in [11] the recorded global state S^* may not have actually occurred instantaneously in the course of computation. However, if the consistent global state detection algorithm started with the system in state S_0 and terminated (i.e., all processes have recorded their states and the states of all incident channels) with the system in state S_ϕ then we want to prove that the recorded global state S^* is reachable from S_0, and S_ϕ is reachable from S^*.

Theorem 3.5

There exists a sequence of computations (η_1,η_2) such that:

$$S_0 \xrightarrow{\eta_1} S^* \xrightarrow{\eta_2} S_\phi.$$

Proof: Without loss of generality, assume the sequence of events that lead the system from instantaneous global state S_0 to final instantaneous global state S_ϕ be $(e_1,e_2,\ldots e_n)$. This sequence represents a serialized schedule of events so that e_i preceding e_j in the list implies that the occurrence of e_i is not later than that of e_j.

We wish to show that $(e_1,e_2,...e_n)$ could be mapped (1-1 and onto) into $(e_1',...e_g',e_{g+1}',...e_n')$. So that $S_0 \xrightarrow{(e_1',...e_g')} S^* \xrightarrow{(e_{g+1}',....e_n')} S_\phi$, and that $(e_1',...e_g')$ and $(e_{g+1}',....e_n')$ are valid schedules of events in the sense that causality constraint is satisfied.

(a) Existence of $(e_1',...e_g')$:

From theorem 3.4, a recorded Consistent global state S^* contains process states associated with a cut containing only forward edges. The resulting S partition of the event graph is acyclic and contains a subset E_s of $(e_1,...e_n)$. We could generate $(e_1',...e_g')$ by traversing exactly those events in E_s so that e_i' precedes e_j' in the list only if e_i' can reach e_j' in the event graph or they are mutually unreachable. This corresponds to generating a one-processor list schedule of the events in E_s. This list, $(e_1',...e_g')$, clearly leads the system to S^*.

(b) Existence of $(e_{g+1}',....e_n')$:

The events in the T-partition of the resulting event graph are acyclic because of the causality constraint. So similar to (a), a valid one-processor list schedule of these events can be deduced leading the system from S^* to S_ϕ.

Q.E.D.

Underlying the reachability property elaborated in theorem 3.5 are two other important properties useful in applications. They are Recoverability and Global-invariant preservability.

3.3.1.1 Recoverability

Definition:

- A distributed system is *functional* if for every execution of the system starting from the same initial state, the same sequence of events and computation occurs in each of the processes.
- In a (error free functional) distributed system if during re-execution after rollback of all processes and channels to states corresponding to the cut representing a recorded global state, the same sequence of events reoccur in each process as they had occurred before rollback, then the recorded global state is defined as a *recoverable state.*

From the above definitions, it can be observed that a recoverable global state is useful in rollback recovery of a fault-tolerant distributed system [25], assuming the system to be functional.

Lemma 3.3: A Consistent global state S^* is recoverable in a functional system.

Proof: The result is trivially inferred from theorem 3.5. Since $S_0 \xrightarrow{\eta_1} S^* \xrightarrow{\eta_2} S_\phi$, the events in η_2 are exactly preserved when the system rolls back to S^*, under functional behavior of the system.

Q.E.D.

Theorem 3.6

A Stable global state is not necessarily a recoverable state.

Proof: Consider the two-process example in fig.3.4. The cut XX′ yields a Stable global state. Let the system be rolled back to this state i.e. P_1 rolls back to s_1 and P_2 rolls back to s_2. During re-execution P_1 will re-execute e_1 and e_3 sending out messages M_1 and M_2 again. However, P_2 will only re-execute e_4 and thus will receive M_1 (erroneously) while the correct matching receive event for e_3 is not executed in P_2.

Q.E.D.

3.3.1.2 Global Invariant Preservability

The distributed system performs a computation represented by an event graph. The actual timing of these events is not crucial but the precedence relationship (causality) among matching send and receive events must be obeyed. We have interpreted this precedence graph by a serial uniprocessor schedule of these events so that there are many uniprocessor schedules for a given precedence graph.

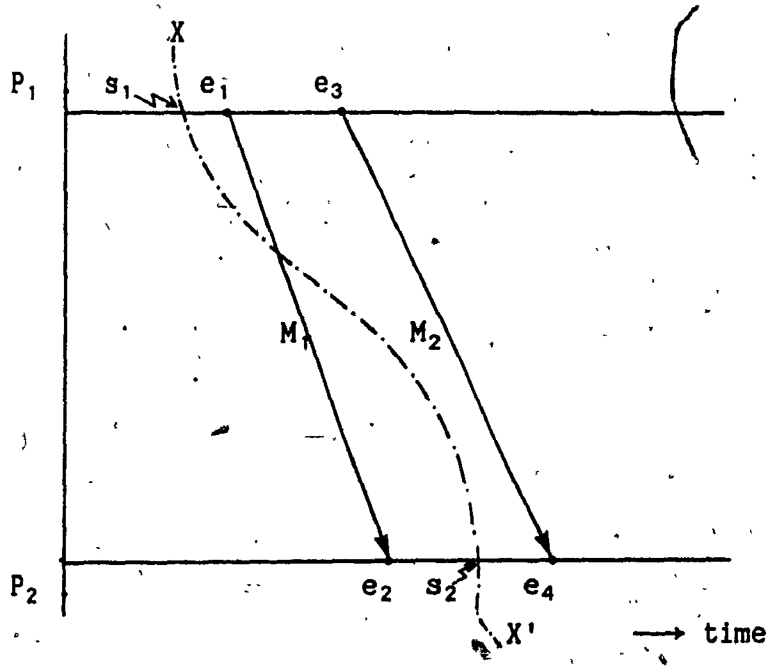


Figure 3.4: A distributed computation

Definition

A global invariant of a distributed system is a property satisfied by the processes as long as the computation (sequence of events) that actually occurs corresponds to a valid uniprocessor schedule representing the given precedence graph of events.

Lemma 3.4: A Consistent global state S^* preserves a global invariant, i.e., the global invariant holds also in a Consistent global state.

Proof: Since a Consistent global state S^* is reachable from S_0 via C_1 which is a valid uniprocessor schedule (Theorem 3.3), S^* preserves a global invariant.

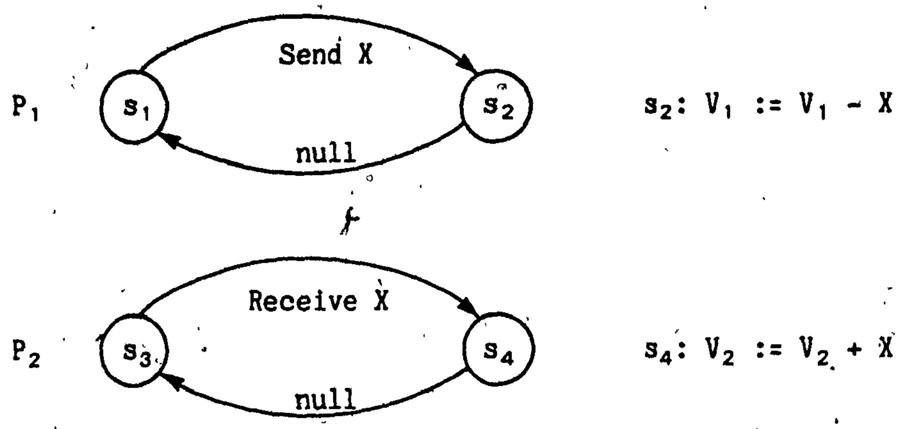
Q.E.D.

Theorem 3.7

A Stable global state does not necessarily preserve a global invariant.

Proof: Consider the following two-process system with a global invariant $V_1 + V_2 = n$ (a constant). The state diagram of the two processes are shown in fig.3.5a and the S-T graph in fig.3.5b. In the stable cut marked YY′, the recorded stable state yields for the global invariant

$$\begin{aligned} V_1 + V_2 + X \text{ in transit} &= V_1(\text{initial}) + (V_2(\text{initial}) + X_1) + X_1 \\ &= \{V_1(\text{initial}) + V_2(\text{initial})\} + 2 * X_1 \end{aligned}$$



Global Invariant:

$V_1 + V_2 +$ Value of X in message in transit if any = n (a constant).

Figure 3.5a: State diagram

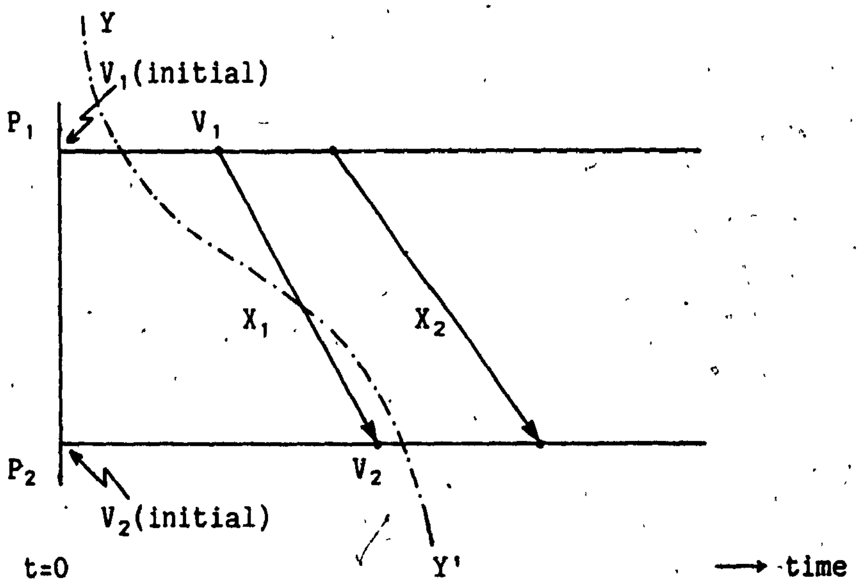


Figure 3.5b: Computation preserving Global Invariant

$$\neq V_1(\text{initial}) + V_2(\text{initial})$$

$$\Rightarrow \text{violation of global invariant.}$$

Q.E.D.

Global invariants are useful in system diagnosis and fault-tolerant computing. The above theorems reveal the use of consistent global state in such applications.

3.4 Synchronized Global State

Applications such as domino-free rollback and recovery [38] and checkpointing in distributed databases [18] require a tight synchronization of local recording of process states to make the latter meaningful for the applications. In this context the notion of Synchronized global state is introduced.

Definition: The recorded states of two processes, P_1 and P_2, are mutually synchronized if

(CS1): All messages sent by P_1 to P_2 prior to the state recording of P_1 have been received by P_2 before P_2 records its state, and vice versa.

(CS2): The recorded state of P_2 does not depend on any message sent by P_1 after P_1 has recorded its state, and vice versa.

A set of recorded states of two or more processes form a synchronized global state if they are pair wise mutually synchronized to one another.

Theorem 3.8

The recorded global state corresponding to a synchronized cut must constitute a Synchronized global state.

Proof: The satisfaction of (CS1) by any arbitrary process pair P_i and P_j implies that there is no forward channel edge crossing the part of the cut between P_i and P_j. Similarly that of (CS2) implies the absence of backward channel edges. Thus the theorem.

Q.E.D.

Lemma 3.5: The recorded global state corresponding to a consistent cut does not necessarily form a Synchronized global state.

Proof: This follows from the fact that a consistent cut may contain forward channel edges and thus the recorded global state may violate (CS1).

Q.E.D.

3.4.1 Applications of Synchronized global state

Synchronized global states have been used to evolve schemes for domino-free rollback recovery of distributed systems [38]. In order to eliminate the possibility of domino rollback, the processes are synchronized at some specific predetermined points in their computation to perform state recording. An important side-effect of this scheme is that interprocess messages are not saved in the global state because of the cause-effect synchronization. However, the performance of the application may suffer because of the freezing required to achieve synchronization as will be explored later.

Synchronized global state ideally matches the requirements of checkpointing in distributed databases. The checkpoints of a distributed database must satisfy the following consistency requirements:

(a) If the effect of a transaction is included in the checkpoint recorded in one site, then the effect of this transaction (if any) must be included in the states recorded in all other sites.

(b) If a transaction T_i depends on another transaction T_j then if the effect of transaction T_i is included in a recorded checkpoint then the effect of T_j should also be included.

It is obvious that the above conditions are reducible to (CS1) and (CS2).

3.5 Relationship between the Global states

To summarize the results presented in this section the following corollary is stated.

Theorem 3.9: A recorded global state satisfies the following relation:

Synchronized global state $\Rightarrow$ Consistent global state,

Consistent global state $\Rightarrow$ Stable global state, and

Stable global state $\Rightarrow$ Statistical global state.

But the relation is not commutative.

Proof: It follows from Theorems 3.1,3.2,3.4,3.6,3.7 and 3.8

Q.E.D.

We are not aware of any earlier work which classifies the useful global states of a distributed system. Various problems requiring global information have been solved independently by researchers and their solutions have been tailored to specific problems. Seminal work in formalizing Consistent global state (as we define it) has been done by Chandy and Lamport [11] even though the notions of consistency and stability are not explicitly distinguished by them.

3.6 Summary

Efficient solutions to a number of problems in distributed systems require gathering state information of the whole system. Based on the requirements of such applications we have identified four types of global states, namely, Statistical global state, Stable global state, Consistent global state and Synchronized global state. These states have been related to four distinct types of cuts in the event graph model of the distributed computation.

Classification of global states has enabled us to focus on generic problems in each category and examine the capabilities of their solutions. Statistical global states and Synchronized global states require only states of processes to be gathered whereas Stable global states and Consistent global states additionally require the knowledge of messages in transit. Essential difference between Stable global state and Consistent global state is in the degree of coordination required between the distributed processes in order to account for the messages in transit. In recording a Statistical global state the information about messages in transit is not essential. However, for a Synchronized global state no messages must be in transit during the state recording phase. Interesting properties of Stable global states like stable property and global minimum detection have been derived. This provides a common basis for solving problems like termination detection, deadlock detection, garbage collection, and updating simulation clocks in discrete event simulation. Consistent global states possess properties like reachability, recoverability and global invariant preservability. Solutions to problems such as rollback recovery make use of Consistent global states. The Statistical global state is used in applications such as task scheduling and load balancing. The Synchronized global state has been used in rollback error recovery, and checkpointing of distributed databases. We have also shown that Synchronized global states $\Rightarrow$ Consistent global states $\Rightarrow$ Stable global states $\Rightarrow$ Statistical global states (but not vice versa).

We have assumed the existence of an initiator processes which start the global state recording, and compiler processes which compose the global picture. There are other models in which explicit initiator or compiler processes may not exist [6,12,31,32]. However, the characterization of global states is equally applicable to these models also.

Chapter 4

Algorithms for detecting global states

Chandy and Lamport [11] have suggested a simple algorithm (CLA) for detecting the Consistent global state correctly. Their algorithm is however inefficient since it requires a test message to be transmitted on every channel of the system. In addition they require the communication subsystem used for message transmission to be lossless and FIFO. Spezialetti and Kearns later integrated the CLA algorithm with a state compilation algorithm [43]. Their compilation phase efficiently disseminates the information obtained in the probe phase.

In this chapter we will first consider a practical model of the communication subsystem which supports Non-FIFO channels. The problem of detecting a Consistent global state when such channels are used will be discussed. A new algorithm for detecting a Consistent global state is then suggested. The algorithm is later simplified to detect Stable global states and Statistical global states.

4.1 Variations of the Working Environment and the Model

Recently Chin and Hwang [14] reported designs of packet switched multi-stage interconnection structures which possess multiple paths between sources and destinations and yield significant reduction in buffer-wait delays. However, unordered delivery of messages might occur in this case and automatic resequencing of the received messages in order to satisfy the FIFO property, when the application does not require FIFO ordering, may offset the performance gain. In general, non-FIFO channels allow simpler and more efficient implementation of the communication subsystem [41,44]. In a related study Kumar [26] reported an increase in message delay by about a factor of three due to resequencing in communication networks. Moreover non-FIFO delivery characteristic could be encountered even in a FIFO channel in the case of loss of a message which induces a subsequent retransmission of just the erroneous message. There are important applications of DCS such as distributed discrete event simulation, and process control involving periodic sensor sampling which do not require the communication channels to be FIFO [5,49]. So non-FIFO communication is a realistic environment for DCS.

Local area networks and close proximity networks are sometimes called "thick-wire" networks because of their high bandwidth, low propagation time and extremely low error rates. Slatzer [41], Linton[30], Nelson [33] and Popek [34] strongly argue that it is impossible to achieve totally reliable communication between endpoints of thick-wire networks even on a perfect communication medium without significant support at the endpoints. In view of this argument they propose to utilize high performance communication subsystems which do not guarantee 100% delivery and add appropriate recovery mechanism only at the endpoints. This is added mostly in the application layer which can take intelligent decisions in order to achieve the desired performance. The Locus OS [34], CMU's RPC [33], Uniform datagram service [30], User datagram protocol [35] have all been designed based on these arguments and have shown significant performance improvements. Along the same vein AT&T has proposed a new packet switching protocol called Fast packet switching which is a "lean best effort" protocol and it expects error recovery procedures at the end-points [51]. All these imply that software layers on top of the communication layer should be capable of handling lossy channels.

Distributed interconnection architectures such as Intel's Cosmic Cube, and ISDN communication architectures have separate channels for "out-of-band" signalling. Some standard LAN interconnections, namely Token bus, Token ring and Ethernets, are all supporting prioritized delivery of messages which lead to non-FIFO message delivery. There is much recent interest in developing algorithms which exploit the strengths of the broadcast medium. Dechter and Kleinrock [15], Levitan [28] have proposed a number of such algorithms and this is being given further impetus by current work in group communication and broadcasting [13,19].

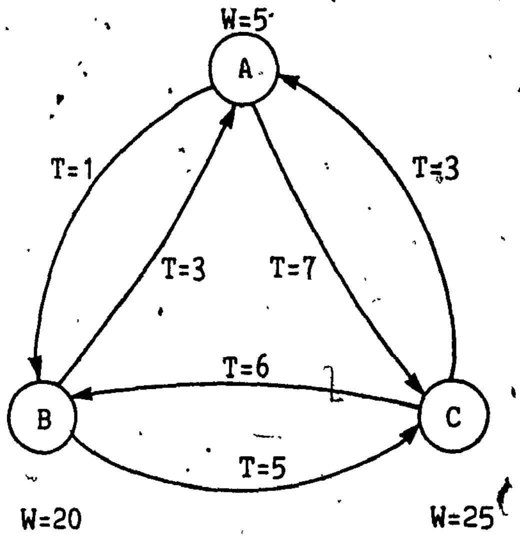
In our opinion important control algorithms like the global state detection algorithm should impose few support requirements on the lower layers and must be able to work using the type of communication support utilized by the application. Such algorithms should also utilize the facilities offered by the communication subsystem to advantage.

The CLA algorithm works only in an environment where the channels are neither non-FIFO nor lossy. It also cannot advantageously utilize either the "out-of-band" signalling or the multicast facilities offered by interconnection networks. A new algorithm (VLR) which functions correctly even in the above environment is presented.

4.2 Conceptual basis for the new Consistent global state detection algorithm

The VLR algorithm is presented through an example. A brief review of the CLA algorithm on the same example is given in Appendix A. Consider the graph of fig.4.1 where the nodes represent rail stations with traffic on the tracks only in the indicated direction. At any given time some wagons will be present in each of the stations and some more will be in transit. Each station is actively engaged in loading wagons and sending them on the output track (if it so desires), or unloading wagons received on its input tracks. Loading and sending, as well as receiving and unloading, are treated as atomic operations.

If the channels are required to be FIFO in the example overtaking of wagons must be prohibited. The variations in the environment (which are not applicable for the CLA) namely allowing the channels to be Non-FIFO, lossy, support out-of-band signalling and multicasting can be interpreted in the context of this example as overtaking of wagons, derailment of wagons, existence of telephone circuits connecting stations,



Legend:

W: Number of Wagons in the station.

T: Number of Wagons in transit.

Figure 4.1: A distributed system

and existence of satellite connections linking the stations respectively.

Suppose station "A" wishes to know the distribution of wagons (global state) in the system. How should "A" proceed? For convenience, assume all wagons are black. Here, the state of a track (channel) is defined as the number of black wagons in transit and the state of a station (process) is the number of black wagons in the stations. Also assume that special "Red" wagons are available and these are used for control purposes. The redwagons correspond to marker messages that are reffered later in the discussion.

The central idea of our algorithm (VLR) is based on the following observation. Consistent global state detection requires coordination of state recording so that if an "effect" is accounted in the global state then the corresponding "cause" should also be included. In other words, a cut of the event graph should not contain backward edges. In order to determine the number of application messages (forward edges crossing the cut) in transit we assign the responsibility of counting the application messages sent out to the sender, and the responsibility of counting the application messages received to the receiver, unlike in the case of the CLA algorithm where counting is done only at the receiver. The algorithm is explained with the same example of fig.4.1. We will interpret a channel state as the number of messages in transit.

In each station observers are permanently assigned to the input and output tracks. In the initial state the wagons are all in the stations, the tracks are all empty and the counters of the observers are set to zero. The output (input) counters are incremented every time a black wagon is sent out (received) on the corresponding output (input) track. Assume that station "A" will initiate the global state recording. Each initiator assigns an unique ordinal number to the successive global state recordings it initiates. For this purpose it maintains a monotonic counter called MKNO. MKNO is initialized to zero when the application is commenced and is incremented every time a new global state recording is started by this initiator. Initiators also maintain a vector (TRANSIT) which contains the number of black wagons in transit on each of the tracks as recorded in its most recent global state recording. TRANSIT like MKNO is initialized to a zero value when the application is commenced. The essence of the VLR algorithm can be described as follows:

Step 1: Station "A" records the number of black wagons currently in the station and also the values of all of its observers (input and output). These observer counters are then reset. Station "A" immediately informs each of the stations "B" and "C" about the new recording either by sending a "red wagon" or by "telephone". All "black wagons" leaving station "A" subsequently will carry a "red flag" labelled with the tuple ⟨A,MKNO⟩. This "red flag" is a redundancy crucial to take care of non-FIFO and lossy environments.

Step 2: Upon receiving the tuple ⟨A,MKNO⟩ station "B" (or "C") (on a red wagon or a black wagon) will compare the received MKNO with that of the last recording initiated by station "A" and if the former value is greater, then this station will execute precisely the same operations as outlined in step 1, except that "red wagons" will neither be sent to station "A" (the initiator) nor to the sender from whom this station has received the tuple.

Step 3: After a station has recorded its state it will forward this information ⟨state, observer values⟩ tagged with the appropriate MKNO to the initiator station "A". Station "A" will consolidate the received state recordings and create the global picture. The number of wagons on each track (state of the channel) is determined by evaluating {Previously recorded TRANSIT of track + new value of observer at Sender - new value of observer at Receiver}. This value is used to update the TRANSIT vector.

The global stae recording algorithm terminates when the application terminates.

4.2.1 Discussion of the VLR algorithm

From the above description we can deduce that the VLR algorithm records a Consistent global state i.e. one in which, if an "effect" is included then the corresponding "cause" is also included, even with non-FIFO and lossy channels. This has been accomplished by the addition of the "red flag" (marker field). If the communication subsystem "assures" reliable end-point connections (FIFO/ non-FIFO) then a significant reduction in the number of marker messages can be obtained. A minimum spanning tree with the initiator as the root can be determined and marker messages sent only on the edges of the tree.

Further reduction of marker traffic can be obtained in communication systems which support broadcasting or multicasting. The n processes can be split into k groups with a control process designated for each group. This could correspond to a situation wherein n processes are assigned over k processors. Assuming equal distributions we have $\lceil n/k \rceil$ processes per group. We further assume that every control process can communicate with all other control processes through a single multicast channel and Marker messages are propagated independently in each group. No process other than the control process will propagate intergroup marker messages. Using this arrangement an initiator in one of the groups sends markers in its group. This marker on reaching the control process of the group will be multicast to all other control processes who in turn will propagate the marker in their own groups. Thus the multicasting features can be effectively used in order to reduce the overheads by sending fewer number of messages.

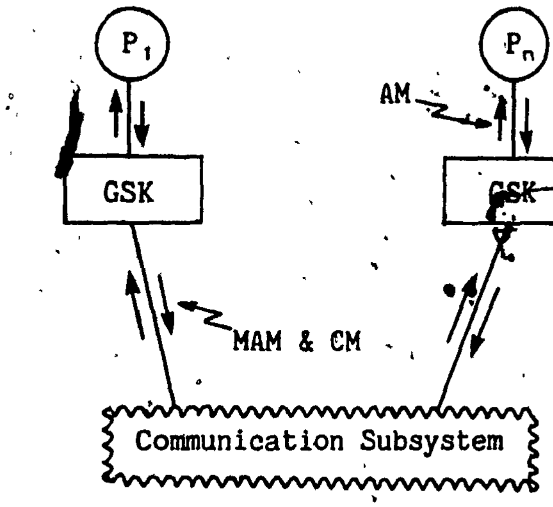
In order to support multiple initiators in the system, VLR associates a separate set of Input/Output counters with each initiator. Marker messages and marker fields will contain a corresponding vector of tuples, one tuple for each initiator so we could envisage the VLR as a re-entrant program with separate data base support for each initiator. Thus the state recordings initiated concurrently by different initiators do not interfere with each other in any way.

Suppose global state recordings are triggered by external conditions. Then, possibly concurrent recordings may be initiated by the same initiator. This situation might occur in applications where a trace of the system state is generated for debugging, error recovery or offline analysis. To accommodate rather than inhibit such situations, the VLR algorithm should be extended. Suppose an initiator P_1 has triggered two global state recordings $MKNO_1$ and $MKNO_2$ concurrently (i.e. one started before the termination of the other), then some process P_a in the system might receive the later recording message (i.e. one with ordinal no. $MKNO_2$) ahead of the first due to the non-FIFO nature of the communication channels. Then the process P_a will record its state whose information is used for (P_1,$MKNO_1$) and (P_1,$MKNO_2$). The receipt of (P_1,$MKNO_1$) later on will simply be skipped as if it was already received. In fact, in general the recorded state will be used as the response for both $MKNO_1$ [illegible] $MKNO_2$ to the initiator. The VLR algorithm with this modification will henceforth be referred to as the Consistent VLR algorithm.

4.2.2 Global State Kernel: An Underlying Issue

The relationship between an application/ system process, the global state detection process(es) and the communication layer may not be apparent at first sight. We adopt the following scheme for presentation, (as well as for our implementation in a DCS testbed). For every process a global state kernel (GSK) is attached as a layer between the application and the communication subsystem as shown in fig.4.2a. The GSK contains the input and output counters. Appropriate marker messages are generated by the GSK layer after recording the state of the process. All application messages being sent out are intercepted by the GSK layer. The GSK layer is responsible for incrementing the appropriate output counters and appending the MKNO vector to the application message before forwarding it to the communication subsystem. On receiving application messages from the communication subsystem, the GSK extracts the MKNO vector, increments the appropriate input counter, performs required state recording and marker generation functions before forwarding a message to the application layer. However, on receiving a marker message only the required state recording and marker generation functions are performed.

Compilation of the global state is done at the GSK kernel of the initiator process. The tasks of local state recording and global state compilation are given in fig.4.2b in the form of pseudo codes.



Legend:

GSK: Global State Kernel

AM : Application Message

CM : Marker Message

MAM: Application Message with Marker vector

Figure 4.2a: Global State Kernel

Nomenclature for the VLR algorithm

Applicable to all processes

I_{ijk} : Counter on input channel$_{ij}$ belonging to the k^{th} initiator

O_{jik} : Counter on output channel$_{ji}$ belonging to the k^{th} initiator

$MKNO_i$: Ordinal number of the latest state recording initiated by an Initiator with Initiator Id.number = i.

Every State recording is tagged with the tuple ⟨d,j,F,L⟩ where,

d: Initiators's Id., j : State recording of process P_j

F,L : First & Last initiations with which this recording is associated.

Applicable to Initiator processes

For every new initiation a frame with the following format is opened

INN	PFC	CFC	PRSTAT	TRANSIT

INN : Ordinal number of initiation

PFC : Status of Previous frame; Initially set to CFC^{INN-1}.

CFC : Status of Current frame; Initially set to 0.

$PRSTAT_j$: Status of process P_j; Initially set to Null.

$TRANSIT_{ij}$: Number of messages in transit on channel$_{ij}$; Initially set to 0.

Significant values for PFC and CFC fields

ALLRECRX : Indicates that for this frame, recorded states from all processes have been received, and TRANSIT contains (No.MsgsSent-No.MsgsRx) for each channel, relative to the previous frame.

COMPLETE : Indicates that for this frame, recorded states from all processes have been received, and TRANSIT contains exact number of messages in transit on the channels.

INITIAL STATE

```
/*     In all processes $P_j$          */
  For all i,j,k set $I_{ijk}$ := 0; $O_{ijk}$ := 0; $MKNO_i$ := 0;
```

Figure 4.2b : Pseudo Code for State detection and Compilation (contd.)

```
**State Recording** code for $P_i$
{
  If process $P_i$ is Initiating Global state detection then
  { /* Let d = Initiator Id.number of $P_i$ */
    $MKNO_d$ := $MKNO_d$ + 1; Open and Initialize frame $MKNO_d$;
    Save and Reset $I_{kid}$, $O_{ijd}$ for all $P_k$, $P_j \in$ (Neighbours of $P_i$);
    Save State of $P_i$; Tag saved values with Id: <d,i,$MKNO_d$,$MKNO_d$>;
    Send Marker messages with marker field set to MKNO vector on all output
       channels of $P_i$; }
  If Marker or Application Message is received by $P_i$ then
    /* Let the message have come from $P_r$ */
  { Extract marker field RMKNO from message;
    TEST := false;
    For m := 1 to Number of Initiators do
     { If $MKNO_m$ < $RMKNO_m$ then
        { Save and Reset $I_{kim}$, $O_{ijm}$ for all $P_k$, $P_j \in$ (Neighbours of $P_i$);
          TEST := true; Save State of $P_i$;
          Tag saved values with Id: <m,i,$MKNO_m$,$RMKNO_m$>
          $MKNO_m$ := $RMKNO_m$;
          Send saved value to Initiator with Id.number = m }}
    If TEST then
     { Send Marker messages with marker field set to MKNO vector on all output
          channels of $P_i$ except to the Initiator [illegible]; }}}
```

```
**State Compilation** code in Initiator with Initiator Id.number = d
/* Invoked on receipt of a state recording message. Let the tag of the
received message be <d,j,F,L> */
  If $PRSTAT^F_j \neq$ NULL then Exit; /* Duplicate */
  For all Output channels of $P_j$ do $TRANSIT^F_{ji}$ := $TRANSIT^F_{ji}$ + Received $O_{jid}$;
  For all Input channels of $P_j$ do $TRANSIT^F_{ij}$ := $TRANSIT^F_{ij}$ - Received $I_{ijd}$;
  For k = F to L do { $PRSTAT^k_j$ := Received State of $P_j$; Update $CFC^k$ ; }
  For all frames k with $PFC^k$ = COMPLETE and $CFC^k$ = ALLRECRX do
      { Set $CFC^k$, $PFC^{k+1}$ to COMPLETE;
        For all channels$_{ij}$ do $TRANSIT^k_{ij}$ := $TRANSIT^k_{ij}$ + $TRANSIT^{k-1}_{ij}$;}
```

Figure 4.2b : Pseudo Code for State detection and Compilation

4.3 Correctness of the algorithms

We will prove the correctness of the Consistent VLR algorithm. Incidentally, we show in Appendix A that the CLA algorithm records a Consistent global state. In the following, a message is said to be in transit if it had been sent out before the sender recorded its state but was not yet received by the time the receiver recorded its state.

Correctness of the Consistent VLR algorithm

In the case of VLR, the communication medium is extended to be non-FIFO. This is modelled as a communication system which possesses nondeterministic delivery ordering properties. An event graph for a distributed computation that has occurred in this environment is shown in fig.4.3. Non-FIFO behavior is characterized by forward edges that cross-over other edges. Lost messages and out-of-band signalling (priority messages) can be treated as special cases of non-FIFO delivery. A lost message is one which is overtaken by all later messages. A priority message overtakes previous messages sent by a process to the same receiver. The Consistent VLR algorithm has been designed for this communication model. The correctness proofs for the Consistent VLR algorithm are given below.

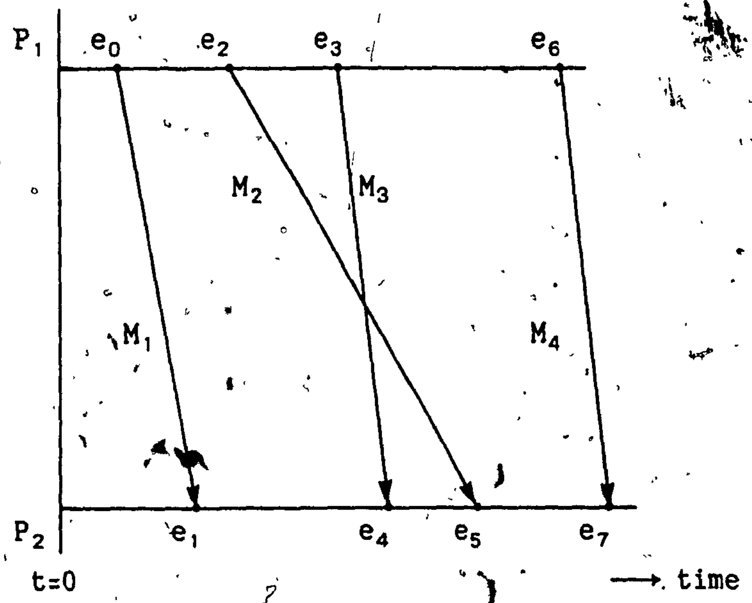


Figure 4.3: A distributed computation in a Non-FIFO environment

Theorem 4.1

The Consistent VLR algorithm always derives a global state associated with a cut without backward edges.

Proof: First we infer, from the program code (fig.4.2b) and previous description, that state recording initiated by distinct processes do not interfere with one another. This is apparent once we realize the Consistent VLR code employs distinct database (counters and marker flags) for distinct initiators. So we can proceed in our proof as if we only have a single initiator to consider.

Case (i): There is only one active recording invoked by the initiator.

Assume the contrary, and (e_c,e_d) an edge from process P_1 to process P_2 be a backward edge recorded in the cut. So the recorded state by Consistent VLR includes some local state s_1 for P_1 and s_2 for P_2 such that s_1 precedes e_c and s_2 succeeds e_d. Since e_c is a post-recording event, the message sent by e_c carries a marker-flag corresponding to this initiator so that when it is received at e_d by P_2, if P_2 has not already recorded its state, P_2 will be forced to do so before accepting the actual message. Thus e_d must be a post-recording event in P_2 (i.e. s_2 precedes e_d) which is a contradiction.

Case (ii): More than one active recording invoked by the same initiator:

Each invocation is identified by $\langle P_i,r\rangle$ where P_i is the initiating process and r the ordinal number of the invocation. Since the program code and database are shared when the same P_i is involved, we need to examine this case. In particular, consider a process P_k receiving $\langle P_i,r\rangle$ while the last marker message containing P_i received by P_k is $\langle P_i,u\rangle$ and $u < r-1$. So the recordings at P_k for invocations u+1,...r-1 and r, are all now collapsed to the same state (of P_k and its channels), according to the program code. We need to prove that this recorded state satisfies the theorem.

We leave the proof for the correct interpretation (compilation) of channel state to lemma 4.1, as our task here is to prove the resulting state recording does not correspond to a cut containing a backward edge (e_c,e_d). This is concluded by observing:

(1) Based on the arguments of case (i) it can be concluded that cut for invocation r does not have any backward edges.

(2) {The pre-recording events for $\langle P_i,u+1\rangle$} $\subseteq$ {that for $\langle P_i,u+2\rangle$} ... $\subseteq$ {that for $\langle P_i,r\rangle$}.

(3) If collapsing the recording for invocation u+1,...r to the same state induced by invocation r leads to a cut for invocation u+i $(u+1 \leq u+i \leq r)$ containing a backward edge, then the same edge will be contained in the cut for invocation r (as illustrated in the fig.4.4) which is a contradiction.

Q.E.D.

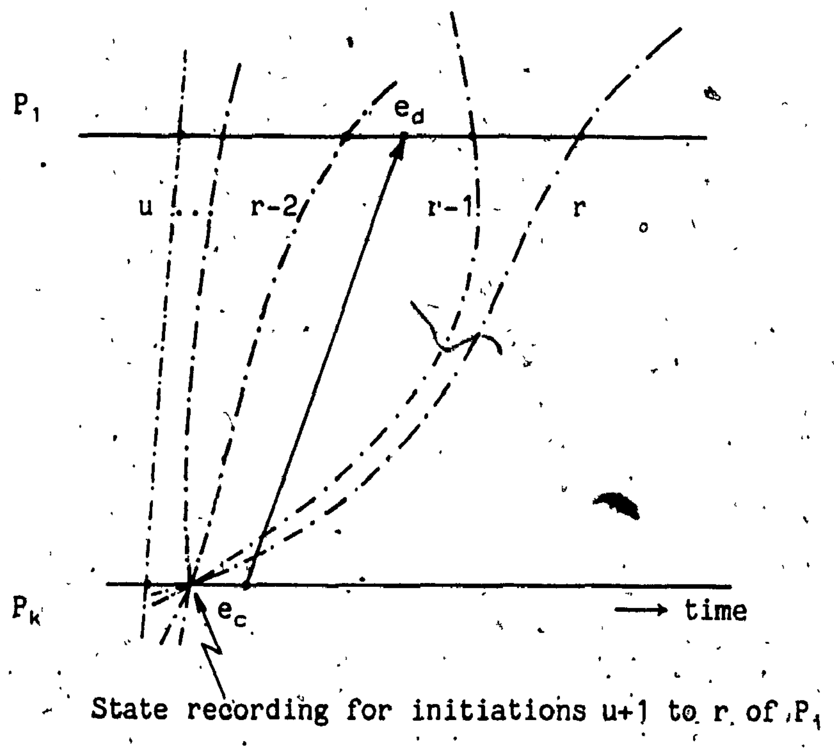


Figure 4.4: Collapsed state recording $\langle P_1, u+1\rangle$ to $\langle P_1, r\rangle$

Lemma 4.1: The local state information can be successfully compiled at an initiator so that the channel state for an invocation would indicate the number of messages supposedly in transit, i.e., sent by a sender before its recording but not yet received by the receiver before the latter records its state.

Proof: The number of messages in transit on a channel from P_i to P_j corresponds to the number of forward edges in the cut (associated with the recorded state) whose send nodes are in P_i and receive nodes are in P_j.

We notice from the program code for each initiator and each channel, $TRANSIT_{ij}^{q}$ is maintained to yield the number of messages in transit on $channel_{ij}$ for the q^{th} recording. This is to be verified. Accordingly, $TRANSIT_{ij}^{q}$ is updated successively for increasing values of q as

$$TRANSIT_{ij}^{q} = TRANSIT_{ij}^{q-1} + O_{ij}^{q} - I_{ij}^{q}.$$

$$TRANSIT_{ij}^{0} = 0$$

where

O_{ij}^{q} = recorded number of messages sent out by P_i between the $(q-1)^{th}$ and the q^{th} recording.

I_{ij}^{q} = recorded number of messages received by P_j between the $(q-1)^{th}$ and the q^{th} recording.

Notice that if P_i or P_j has a collapsed recorded state, for $\langle P_r,q\rangle$ to $\langle P_r,u\rangle$ where $u > q$, recorded when $\langle P_r,u\rangle$ was received ahead of $\langle P_r,q\rangle$ then O_{ij}^k or I_{ij}^k is set to zero for $k = q+1$ to u. Solving the recurrence,

$$\mathrm{TRANSIT}_{ij}^{u} = \sum_{k=0}^{q} O_{ij}^{k} - \sum_{k=0}^{q} I_{ij}^{k}$$

$$= \text{number of messages yet in transit on the channel.}$$

Q.E.D.

4.4 Extensions of the Consistent VLR algorithm

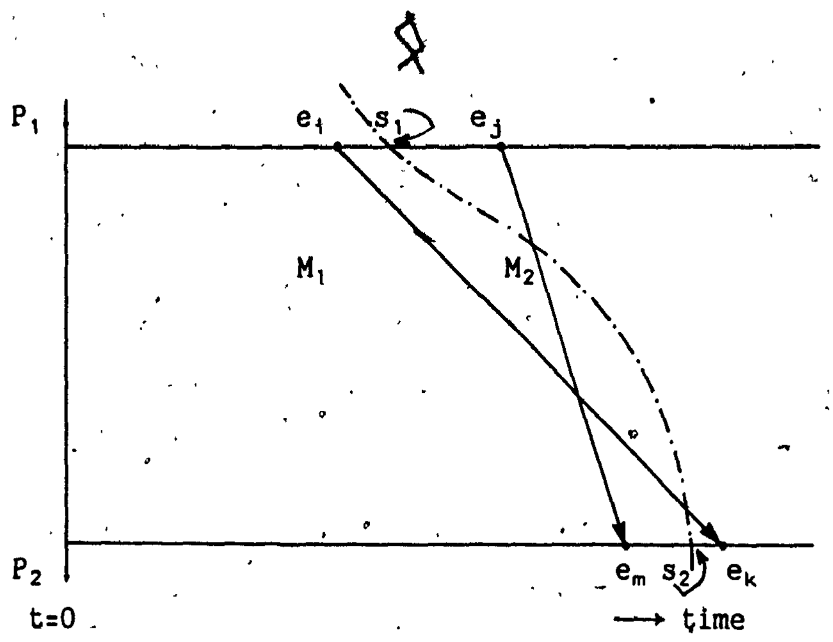
4.4.1 Algorithm for detecting a Stable global state (Stable VLR)

The Consistent VLR algorithm can be simplified in order to detect a Stable global state assuming the channels are reliable. The marker flag field (Red flag) is eliminated from the messages. In addition marker messages (red wagons) are sent only on the edges in a preselected minimum spanning tree of the process graph. When these modifications are incorporated the Consistent VLR algorithm will be called a Stable VLR algorithm.

A noticeable difference in the executions of Consistent VLR and Stable VLR is that some entries in the transit vector might become negative. If an entry $\mathrm{Transit}_{ij}^{q}$ is -X then X messages were sent to P_j after P_i has recorded its state for the q th time and were received by P_j before its state was recorded for the q th time. So there are X backward edges intersecting the cut. As before a positive value for Transit entry shows the number of messages in transit on the corresponding channel.

In a non-FIFO communication environment the simple Stable State VLR algorithm could fail to record the Stable state correctly. As illustrated in fig.4.5, a post recording message received ahead of some pre-recording message will result in a wrong count of the number of messages in transit on the channels. In order to avoid such an effect the pre-recording messages have to be distinguished from the post recording messages. Separate counts will have to be maintained for the pre and post recording messages. In case an initiator is allowed to concurrently initiate a number of iterations, then the post recording messages will have to be differentiated based on their iteration number, and counted separately. In general an infinite set of counters might be required. When the overhead due to large set of counters is quite high, it does not seem to be advantageous to use the Stable State detection algorithm, as the Consistent State algorithm might then be relatively simpler.

However, it should be noted that Stable State VLR has its own areas of applications. Problems like termination detection need to know only whether there are messages in transit and not exactly how many are in transit. Also initiators in this case will not initiate multiple recordings concurrently. In such situations the Stable State algorithm with modification to detect post recording messages would suffice.



State recorded = $\langle s_1, s_2, C_{12}=\emptyset, C_{21}=\emptyset \rangle$

P_2 also records that a post recording message has been received.

Figure 4.5: Stable state detection in a Non-FIFO environment

An interesting application of Stable States is in Distributed discrete event simulation. For this application the Stable state VLR should be modified in order to detect the global minimum. The basis and details of this modification are given in Chapter 5.

4.4.2 Algorithm for detecting a Statistical global state (Statistical VLR)

In a Statistical global state we are interested in information available at the processes like queue length, spare CPU cycles etc. Hence the state of the processes are only of interest and there is no need to keep track of the messages in transit. In view of this the Stable VLR can be modified by discarding the input and output observers to realize an algorithm for detecting the Statistical global state.

4.5 Algorithm for Detecting Synchronized Global State

It is difficult to obtain a safe (deadlock free) and correct algorithm to detect a Synchronized global state without placing some additional restrictions on the interaction between state recording and the subsequent continuation of a process. The example in fig.4.6 illustrates the reasoning involved.

From the example, it is observable that if the interprocess communication is totally unconstrained then it is impossible to obtain a Synchronized cut bounded by the local states marked by A_i, B_i and C_i. Specifically, some forward or backward channel edges must be included,

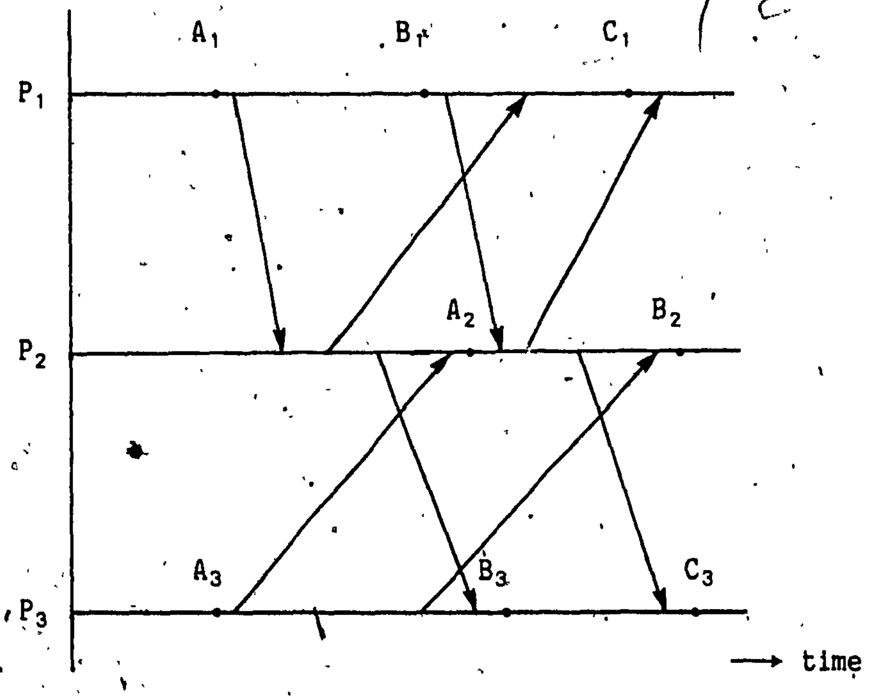


Figure 4.6: A distributed computation with no Synchronized cut

until eventually no further interprocess communications is pending or will arise. We do not propose any new algorithm for detecting Synchronized global states. Instead, a brief review of some existing techniques follows.

The structured Conversation scheme [22,38] enforces very strict control on interprocess communication. Interprocess communication is segmented into what is known as "conversations". All partners who participate in a conversation are known in advance and must all concur to start the conversation. A process cannot be a member of two different conversations at the same time. All partners of a conversation leave it together. No process can join a conversation that is already in progress. A simple algorithm for obtaining a Synchronized global state in this model is to freeze the processes before they enter a new conversation, but allow conversations currently in progress to complete before freezing the participating processes. A process which is not part of any conversation can be frozen at any time. In due course all processes will freeze and the state of the processes can be saved in order to capture a synchronized global state. After state recording, the processes can be synchronously resumed.

The transaction model of distributed processing provides the necessary framework in which the whole distributed system can be brought to an idle state in order to capture the Synchronized global state. A probe phase could inform all the processes that a Synchronized state needs to be captured. When a process receives this directive it stops accepting new transactions (subtransactions), but completes all its ongoing transactions. When a subtransaction is not accepted, automatically the transaction gets aborted. In this manner the system will come to an idle state as soon as each process completes its current transactions. At this point the Synchronized state can be captured. However, Fischer [18] has suggested a non-intrusive scheme for capturing a Synchronized global state (he calls it a Consistent global state) in the transaction model. In his scheme instead of freezing the processes he allows them to process new transactions by creating temporary copies of the database. These temporary copies are integrated to the master only after a synchronized global state is recorded. Son [42] has adapted this scheme for checkpointing distributed databases. The details are beyond the scope of this thesis.

4.6 Performance of the global state detection algorithms

Message complexity and Recording delay are the important performance metrics of global state detection algorithms. On analysis of the VLR algorithms it has been found that for the case of designated initiators and compilers, the Statistical VLR, the Stable VLR and the Consistent VLR have similar performance characteristics. Hence in the following only the performance characteristics of the Consistent VLR algorithm is presented under different communication environments. Also, the performance of the Consistent VLR algorithm is contrasted with that of the CLA algorithm.

4.6.1 Message Complexity of the Probe phase of Consistent VLR

(a) Reliable channels point to point communications

As outlined before, the marker messages will be sent only on edges belonging to a spanning tree rooted at the initiator. Each application message carries a marker (flag). Thus for each initiation, the number of marker messages in a system with n processes and L edges will be equal to (n-1) in Consistent VLR as compared to L in CLA. So Consistent VLR has an improvement factor of O(n) if L = $O(n^2)$.

(b) Reliable channels with multicast support

Assume that the n processes are split into k groups and multicast support is available for intergroup communication as described before. In this general case with one multicast marker message the control processes of all the groups will receive the required marker information. Subsequently, ($\lceil n/k \rceil$ -1) unicast marker messages will have to be propagated within each of the k groups in order to reach all the processes. So the overhead will be (($\lceil n/k \rceil$ -1)*k) unicast marker messages along with one multicast message. Depending on the n/k ratio the overhead will vary from 1 (for n groups: broadcast) to n-1 (for one group: point-to-point). A similar trade-off cannot be achieved in the case of CLA because it requires markers to flow through every channel in a FIFO manner and does not make use of the broadcast/multicast facilities.

(c) Lossy Channels

Marker messages might get lost in lossy channels. In this context the initiator after initiating a global state recording could initialize a timer. It would expect to receive the local state recordings from all the processes before the timer times out. If it does not receive recordings from some processes, it could subsequently interrogate them in order to get the required information. For such a solution the markers need still be sent only on the edges of the Minimum Spanning tree. Hence it inherits the same O(n) message complexity. However, each process can introduce redundancy by sending marker messages on all of its outgoing channels instead of just on the Minimum Spanning tree. This redundancy could help in reducing the number of individual interrogations stated above. An upperbound on the number of marker messages in such a case is L i.e. one marker message on each channel.

4.6.2 Message complexity of the Compilation phase of Consistent VLR

This is the phase in which processes send the recorded local state back to the initiator. If we make the same assumption of channels being bidirectional made by Spezialetti [43] then as in their algorithm the recorded states can be sent back using the edges of the spanning tree. In this case the performance for the Compilation phase of global states by NINIT initiators will be $O(NINIT * n^2)$ which is the same as that quoted in [43] for the efficient version of CLA.

4.6.3 Recording delay of Consistent VLR

In the Consistent VLR algorithm a process on receiving the first marker message of a new iteration will record its state and that of all its input and output observers. As this constitutes all the information needed from each process in order to compose the Consistent global state, the complete local recording occurs right away without any waiting. However, in the CLA algorithm the local recording in each process, which commences on receiving the first marker of a new iteration, is completed only after receiving marker messages of the same iteration on all its other input channels. Thus the CLA algorithm incurs additional bookkeeping overheads in contrast with the VLR in environments where an initiator starts multiple iterations of global state recording concurrently.

4.7 Summary

The general and widely applicable model in which processes are designated for initiating and compiling global state recordings has been used here for developing algorithms for global state detection. In this environment a message efficient algorithm for capturing the Consistent global state (Consistent VLR) has been developed. A significant aspect of this algorithm is its ability to retain pertinent information at the sender's site of a communication channel instead of discarding it and recreating the same at the receiver's site. This allows significant reduction in number of control messages propagated in each iteration of the algorithm. Additionally, the local recordings in each process occurs immediately upon receiving the first marker of a particular iteration. This reduces the book keeping overheads when initiators are allowed to start multiple iterations of global state recordings concurrently. Recently, Lai [27] has reported a similar algorithm with similar characteristics for Consistent global state detection. However, this has been reported in the context of recognizing stable properties.

The Consistent VLR is methodically simplified in order to derive algorithms for satisfying the lesser requirements imposed by Stable global state detection and Statistical global state detection. In the general model adopted here there is no significant performance advantage for either the Stable VLR or the Statistical VLR over the Consistent VLR. However, models tailored to specific problems exist in which Stable and Statistical global state detection performs much better than Consistent global state detection. Some of the problems that fit into this category are: Distributed garbage collection [31], Detection of termination distributively [32], Load balancing in distributed systems [53].

No safe algorithm exists for detecting a Synchronized global state in the general model of a distributed system. However, if either the Conversation model or the Transaction model is adopted then simple algorithms for Synchronized global state detection can be evolved.

Chapter 5

Discrete Event Simulation

Updating the simulation clock is one of the main problems of distributed discrete event simulation. The clock updating requires the detection of the event with the earliest simulation time among events distributed in the system. Then the clocks of all the processes are updated to this time. As outlined in §3.2 the Global Minimum detection property of Stable global states is useful in this context. However, in order to minimize the overhead involved in updating the clocks, an integrated approach for detection and compilation of global states will be devised. A brief review of the problem of performing discrete event simulation in a distributed system is first presented to reveal the basic underlying assumptions and issues that arise. The latter is then formulated as three optimization problems to be formally tackled.

5.1 Simulation Environment

A distributed computing system consisting of loosely coupled processors which interact through message passing constitute the distributed event simulator. We can model the simulator by a graph whose nodes represent the processes and whose edges represent the channels through which message communication can be accomplished. In general, a process graph may contain a number of strongly connected components.

In discrete event simulation each process is characterized by a set of functions it carries out. The basic activity pursued by a process is simulation of events. Events are represented by a tuple (Id. of function, invocation-time). The tuple specifies the function of the process that should be executed in order to simulate the event and the time at which the event has to be simulated in the process. The assumptions of the model are as stated below:

(SA.1): Each process maintains a local "simulation" clock. At a particular time, it invokes (i.e. simulates) those events which are on its event list and whose invocation time matches with the local time. We assume the exact time needed to simulate an event is irrelevant, and consider an invocation of an event as an atomic action.

(SA.2): In completing the invocation of an event (and removing it from the local event list), new events of the form (event_i, invoke-time_i) may be created. Such events may be simulated in the same process or in an adjacent process, depending on the simulation application. In the former case, the event is appended to the local event list. In the latter case, it is sent to the target process via a message.

(SA.3): We assume that the system is causal: the invocation time of a created event must be later than that of the event that creates it.

(SA.4): The channels among processes are point-to-point and directed. The delay for transmitting a message through a channel is unknown and unbounded.

The illustration in fig.5.1 shows a simulator that consists of six distributed processes whose local clocks are synchronized to be 20. During simulation, new events may be appended to the local event lists or transmitted to a neighbor process.

5.2 Clock Update Problem

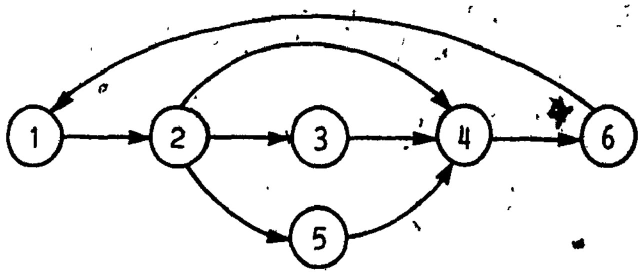
The simulator functions correctly if and only if each process (P_i) never advances its local time (L_i) beyond the invocation time of any event in its event list or that of any event yet to be received on any of its input channels from its neighboring processes.

The problem being addressed is two-fold:

(a) How can the above requirement on clock update be met in the distributed environment?

(b) In what ways can the effectiveness and efficiency of such a solution be evaluated; are there optimal solutions?

Asynchronous (Time Incrementation [10]) and Synchronous (Time Acceleration [8]) algorithms have been developed in order to update the local simulation clocks. In strongly connected components of a process graph the magnitude of time increments that can be obtained in each iteration of the Time Incrementation algorithm is limited by the presence of feedback loops. Hence, Bryant proposed the Time Acceleration algorithm. As the focus here is on advancement of clocks in strongly connected components of process graphs we will be concerned only with



Event List

Node	event	invoke time
1	e_{11}	100
	e_{12}	200
2	e_{21}	150
	e_{22}	160
	e_{23}	170
3	e_{31}	60
4	e_{41}	90
	e_{42}	100
5	e_{51}	65
6	e_{61}	110
	e_{62}	140

Events in transit

Edge	event	invoke time
1,2	e_{13}	70
	e_{14}	120
2,3	e_{24}	110
2,4	e_{25}	120
3,4	–	–
4,5	e_{43}	110
	e_{44}	120
4,6	–	–
5,2	e_{52}	100
6,1	e_{63}	150

Local times at all nodes = 20 units.

Figure 5.1: Process graph 1

the Time Acceleration algorithm.

In the logically synchronous approach for clock update proposed by Bryant, a global "state" detection phase is executed first. The global state in this problem corresponds to determining the earliest invocation time of any event in any event list, or in transit on some channel. To ensure correctness, a process can participate in this global state detection only after it has completed the invocation of all events in its event list whose invocation time coincides with the local time. After this global state detection, a consensus can be reached so that all processes will be notified to update their local time to a common value, thus the notion of logical synchronism. The direct correspondence between the generalized problem of Global minimum detection and this problem is evident. However, the solution proposed by Bryant actually constructs a Consistent global state by flushing all the channels. In addition its Compilation phase is very message inefficient and appears to be ad hoc. Here the formal requirement of the problem is addressed which is then formulated as optimization problems that can be systematically solved, using a variety of algorithm techniques to suit different optimization objectives. The complexities of these optimization problems are analyzed and it will be proved that one of them has a simple optimal solution while the other is NP-Complete for which an efficient heuristic is proposed.

5.3 Formulation of the clock update problem

A distinction between Bryant's approach and ours exists which makes the optimization problem different. Bryant assumes that for obtaining the channel states while constructing the global picture, every channel must be traversed with a test message in order to account for the events-in-transit on that channel. We simplify this requirement by observing that each process is fully aware of the set of events it has created and sent on an output channel to a neighboring process since the last iteration (of the clock update). So if a process P_i maintains a $MINTIM_{ij}$ for its output channel to process P_j which reflects the earliest next event invocation time of the events sent on that channel since the last update, a global state can be pieced together by simply visiting all processes, rather than flushing all channels.

Additional Assumptions

(SA.5): Each process maintains a variable $MINTIM_{ij}$ for each of its output channels. This variable is set to the earliest invocation time of the events sent on that channel after the previous clock update (iteration).

(SA.6): A process is ready to participate in the i^{th} iteration of the global clock update only if:

(i) it has completed the events on its event list whose invocation time coincides with the current time, and

(ii) all the events created by neighbor processes and sent to this process at the $(i-1)^{th}$ iteration have been received.

The latter condition is imposed so that $MINTIM_{ij}$ is confined to the most recent set of events created and sent on each channel. In the Stable global state that gets recorded, the states of the processes correspond to the earliest event currently in the respective event lists and the channel states are abstracted by the MINTIM variables maintained for each of the channels.

5.4 A Conceptual Solution

Imagine one of the processes, say P_1, is chosen as the coordinator for time advancement. A set of inspectors, which we call red tokens and green tokens, are sent out from P_1. The red tokens represent test messages and the green tokens represent marker messages. The exact significance of these messages is discussed later on. The tokens can traverse any edge in a group (counted as one unit cost corresponding to a composite test-message) or in a succession of group(s) (repeated traversal of the same edge by test-messages). A feasible solution of the global state detection problem is obtained when

(CS1): each node (process) has been visited by a red token,

(CS2): all red tokens have returned to P_1, and

(CS3): every edge has been traversed by either a red token or a green token.

Notice that green tokens do not have to return to P_1. The condition CS1 is required to pick up the local information in each process, and CS2 ensures that all local information has returned to the coordinator. The condition CS3 ensures that the untraversed channels are flushed before the next iteration of clock update proceeds (SA.6(ii)).

The above solution contrasts with Bryant's solution which requires that

(BS1): every edge be traversed by a red token, and

(BS2): all red tokens return to the coordinator.

The traversal of an edge by a group of red tokens is interpreted as the transmission of a test message to invoke the receiver process to participate in the clock update operation. The traversal of an edge by a green token serves to flush the channel and also separate event messages generated in successive iterations.

The coordinator starts the i^{th} clock update iteration after completing the required simulation of the $(i-1)^{th}$ iteration and having received all required marker messages corresponding to the $(i-1)^{th}$ iteration. It determines the earliest invocation time among the events in its event list, and communicates this time in the form of test messages. A process will hold the test message for the i^{th} iteration until it is ready to perform the assessment for clock update (SA.6). During assessment the process determines the earliest invocation time t_{new} from the set of events currently in its event list, all those events which it has sent to its neighbors in the simulation iteration just concluded, and the clock value in the received test message. t_{new} replaces the possible global clock value in the received test message which is then forwarded to the chosen neighboring process(es) and it sends marker messages to its other neighbors. Notice that a process does not forward a test message until it has completed its current simulation iteration.

After all the test messages of the i^{th} iteration return to the coordinator, the coordinator determines the earliest invocation time t_{min} among the clock values contained in the test messages. This value t_{min} is then propagated to all processes for updating their local clocks (by sending Set messages).

5.5 Optimization problems

The above formulation yields a feasible solution, but we could select better solutions by asserting certain objective functions that intuitively optimize certain performance aspects. The following are three candidate objective functions:

(Obj1): Minimize the total cost of the solution where cost is the number of edge traversals (with repetitions if necessary) by red and green tokens singly or in groups.

(Obj2): Minimize the total cost of the solution where cost is the number of edge traversals (with repetitions) by red tokens singly or in groups.

(Obj3): Minimize the longest walk by any red token from the coordinator to itself.

Based on the interpretation of the red and green tokens the objective functions Obj1, Obj2 and Obj3 can be visualized as follows. Obj1 optimizes the total number of test and marker messages and thus optimizes the message overhead. Obj2 optimizes the total number of test messages assuming the marker messages are inexpensive. Observe that all test messages of a particular iteration will have to be propagated before the quanta of update can be determined for that iteration. However, the markers need to arrive at their destinations only before the assessment for the subsequent update iteration. So the test messages can be accorded a higher priority. In this context Obj2 is meaningful as the marker messages can be ignored in the optimization criteria. Obj3 optimizes the longest path delay. In a system where the edges in the logical model are realized physically by non-multiplexed channels, the longest sequence of processes which a test message has to traverse in a sequential fashion will determine the rate at which updates can be performed. Hence, Obj3 by optimizing the longest path would lead to a solution that will allow the update to be performed fast.

To illustrate these different objectives and solutions, consider the example process graph depicted in fig.5.2a where process 1 is assumed to be the coordinator. The optimal solution that minimizes the test message overhead that is Obj2 is shown in fig.5.2b. Similarly, those for Obj1 and Obj3 are shown in fig.5.2c and fig.5.2d respectively. Notice that we can abstractly represent the flow of test messages (red tokens) from the

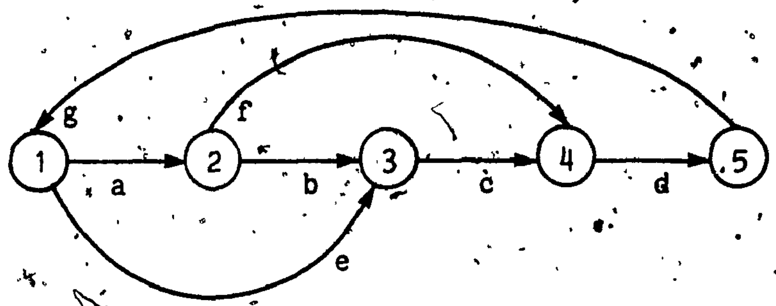


Figure 5.2a: Process graph 2

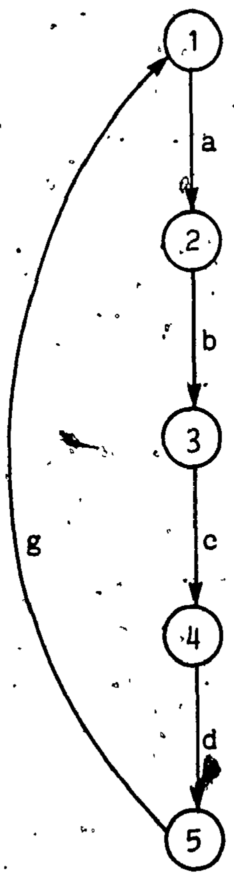


Fig.5.2b: Test Message only.

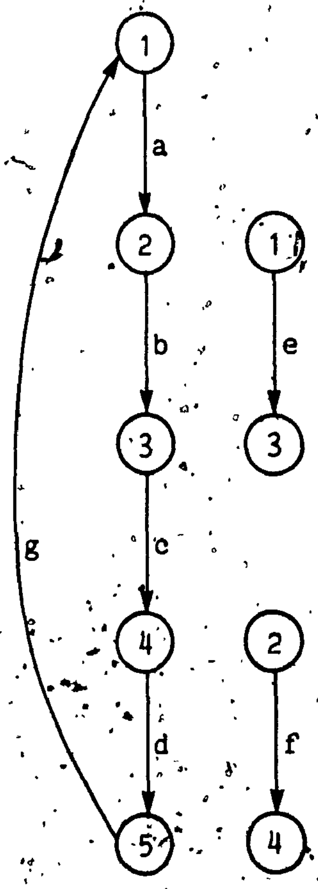


Fig.5.2c: Test + Marker Messages.

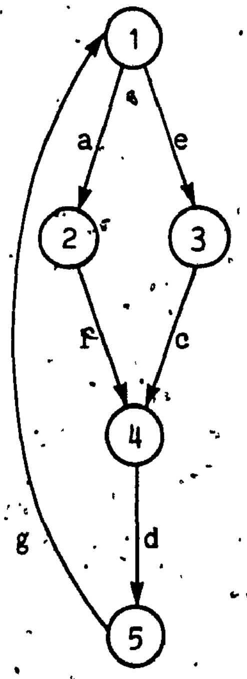


Fig.5.2d: Minimize Longest Walk

coordinator through all processes and back to the coordinator by a suitable regular expression. For example, in fig.5.2d, the solution (af + ec)dg corresponds to the parallel traversal of af and ec followed by dg. Indeed, we can view "+" as the fork (parallel traversal) of two test-message sequences and "( )" as the join or merge of two or more test-message sequences into one. Concatenation of edges represents sequential traversal of these edges, as depicted in fig.5.2e.

5.6 An Optimal Algorithm for Obj3

Algorithm Fast-Update

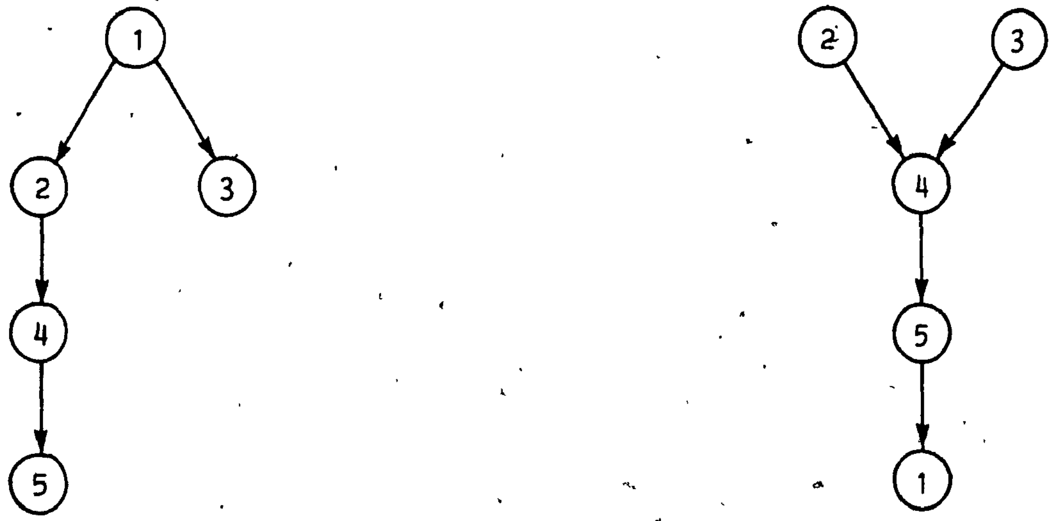
```
begin
    { Obtain a breadth-first-search tree rooted away from the
      coordinator;
      Send test messages to all nodes according to this tree; }
    { Obtain a breadth-first-search tree rooted toward the coordinator;
      Return test message to the coordinator according to this tree;}
end.
```

For the example in fig.5.2a, we will obtain from Fast-Update the two trees shown in fig.5.3, which when merged will yield the solution in fig.5.2d.

Optimal Soln.: Obj2 = abcdg	Optimal Soln.: Obj1 = abcdg ; (Test) and e; f. (Marker)	Optimal Soln.: Obj3 = (af + ec)dg

+	:	fork of two test-message sequence
( )	:	join of two or more test-message sequence

Figure 5.2e: Optimal traversals for Obj1, Obj2 and Obj3



Forward breadth first
Search tree

Reverse breadth first
Search tree

Figure 5.3: FastUpdate search trees.

Theorem 5.1

Algorithm Fast-Update yields an optimal solution for Obj3.

Proof: The first breadth-first search tree yields the shortest path from the coordinator to each process while the second yields the shortest path in the reverse. The two shortest paths together allow the coordinator to pick up the information from the processes in the fastest way possible. Thus the combined solution minimizes the path through each process and so Obj3.

Q.E.D.

Notice that the combining of the two trees may lead to merging of test-messages and further economy of the total number of test-messages. For the example of fig.5.2, only 6 test-messages (a,f,e,c,d,g) are needed, rather than eight (in general 2*(n-1), where n = number of processes). However, this latter aspect is unrelated to Obj3 and will be left to later consideration of Obj1 and Obj2.

5.7 Complexity of Obj1

The complexity of the optimization problem which minimizes the total number of edge traversals by test and marker messages (Obj1) can be revealed by proving the NP-Completeness of the following formulation:

TC: Test-Message Cover

Input: A process graph of n nodes one of which is the coordinator, and a positive integer k.

Output: Yes, iff there exists a routing of test messages from the coordinator which covers all nodes and returns to the coordinator such that the total number of repeated traversal of the edges by test messages is k.

Explanation: We observe that to minimize the total number of test and marker messages is the same as to minimize the total number of repeated traversal of edges by test messages, as those edges not traversed by a test message have to be flushed by a marker message to prepare them for the next iteration, as explained previously.

We prove TC is NP-Complete by reducing the set cover problem [21] to TC. The set cover problem is stated as follows:

SC: Set Cover

Input: A set Q of subsets $\{Q_1,...Q_n\}$ of a finite set $S = \{X_1,...X_m\}$ and a positive integer $k \leq n$.

Output: Yes, iff Q contains a subset Q' so that $\bigcup_{Q_i \in Q'} Q_i = S$ and $|Q'| = k$.

To derive the proof, the following polynomial-time transformation procedure is developed to transform a given instance of SC into an instance of TC.

Problem Transformation:

1. For each $Q_i \in Q$ construct a rooted subtree of the form shown in fig.5.4a. The root is labelled Q_i, and has a single successor D_i. In turn D_i has a set of successors, one for each of the elements in Q_i, which form the leaves of the subtree and are labelled accordingly. In addition, an extra successor labelled T_i is added for D_i.
2. Add a root R whose successors are precisely $Q= \{Q_1,...Q_n\}$.
3. Form a process graph G by adding the following edges:
   (a) For each leaf node labelled with X_j, if $X_j \in Q_i$, edge (X_j, Q_i) is created.
   (b) For each T_i, edge (T_i,R) is created.

For the example, the resulting process graph is shown in fig.5.4b.

Theorem 5.2

A given instance of the set cover (SC) problem of size k has an affirmative output iff the corresponding instance of TC obtained above, possesses an affirmative solution with $\underline{k}$ repeated edge traversals.

Proof:

$(\rightarrow)$: Given a solution of TC with k repeated edge traversals by test messages.

Let $Q_1 = \{X_1, X_2\}$ $Q_2 = \{X_2, X_3\}$ $Q_3 = \{X_1, X_3\}$

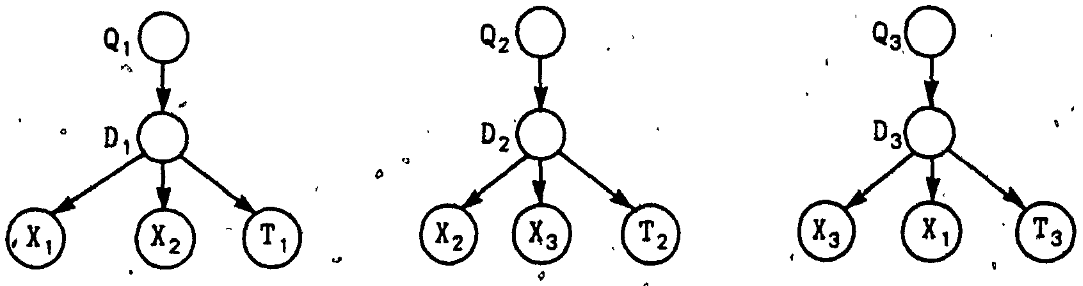


Figure 5.4a: Transformation Step 1.

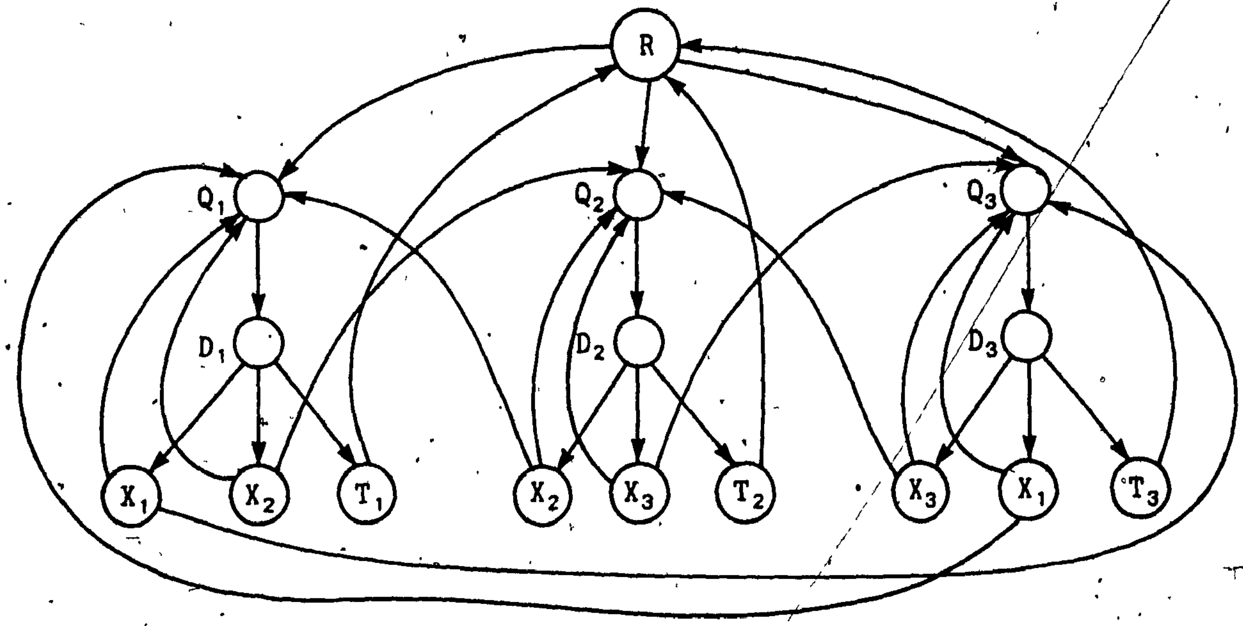


Figure 5.4b: End of transformation.

Let the number of (Q_i,D_i) edges repeatedly traversed by some test messages be $k' \leq k$. Without loss of generality, assume these repeated edges are precisely (Q_1,D_1), (Q_2,D_2)... $(Q_{k'},D_{k'})$. We claim $Q' = \{Q_1,...Q_{k'}\}$ form a set cover of S. To prove this claim, we use contradiction. Suppose the claim is not true so that there exists $t \in S$ and none of the leaves in Q' is labelled with t. Further, without loss of generality, let $Y = \{Q_{k'+1},...Q_{k'+r}\}$ and $\forall\ Q_i \in Y$ let $t \in Q_i$. Now the leaf nodes labelled with t in Y can only pass the return test messages to some Q_i in Y. There are r such leaf nodes and r such Q_i. Either (a) by the pigeon hole principle, some $Q_i \in Y$ is visited twice and consequently (Q_i,D_i) visited twice, or (b) the traversal of the t-leaves form a cycle without repeating Q_i's. In case (a), it contradicts the assumption (then $Q_i \in Q'$). In case (b), the t-leaves cannot return a test message to R which routes through the cycle once and stops. Thus Q' must form a set cover of size $k' \leq k$; a set cover of size k thus exists.

($\leftarrow$): The reverse is straight forward. Given a set cover, say $Q' = \{Q_1,...Q_k\}$. R sends test messages to all $Q_i \in Q$, which route them to all leaves except the T_i's. These leaves then return test messages to nodes in Q' which then forward them through the corresponding T_i's back to R. The number of repeated edge traversal that could occur only at (Q_i,D_i)'s by test messages is precisely k.

Q.E.D.

Corollary 5.1: TC is NP-Complete.

5.8 Heuristic Solution

Having identified the NP-Completeness of the optimization problem, an efficient heuristic solution is justifiable to achieve the objective of minimizing the number of messages used in each update iteration. The heuristic proposed here consists of a two phase process. In phase-I, the algorithm creates a forward breadth-first spanning tree rooted from the coordinator. This tree is used to broadcast test messages which will reach all nodes in the graph G. Subsequently in phase-II, reverse paths from the leaves are derived greedily from a reverse breadth-first spanning tree rooted to the coordinator. Greediness is maintained so that the return path, of a leaf identified in phase-I, invokes the minimal marginal cost measured in terms of the additional edges which have to be repeated. It will be proved that an edge will be traversed at most twice by test messages in an iteration of clock update.

Definition:

For each node v in a tree, d(v) = distance of v from the root of the tree.

5.8.1 Algorithm MINMES:

Phase I:

Step 1: Obtain a forward breadth-first spanning tree BFT_f of G rooted from the coordinator R. Partition the set of leaves as:

T: Subset of leaves which are directly adjacent to R, and

N: the remaining subset of leaves.

Label each edge in G which is found in BFT_f by the set of leaves reachable from the root R via that edge in BFT_f. For the process graph of fig.5.5a this labelling scheme is exemplified in fig.5.5b. We assume that each edge in G has two buckets to hold labels, and in this step all labels assigned to an edge are placed in the first bucket.

Phase II:

Step 2: Obtain a reverse breadth first spanning tree BFT_r of G rooted to the coordinator R as exemplified in fig.5.5c.

Step 3: For each $t \in T$ (leaves in BFT_f directly adjacent to R), remove (t,R) from BFT_r and label (t,R) in G by t' (placing t' in the first bucket of (t,R)), as shown in fig.5.5d. A forest may result for BFT_r.

Step 4: Choose and remove an $x \in N$ (leaves which must return test messages to R). Find among its direct successors Succ(x) the successor y whose d(y) is minimum. Label the

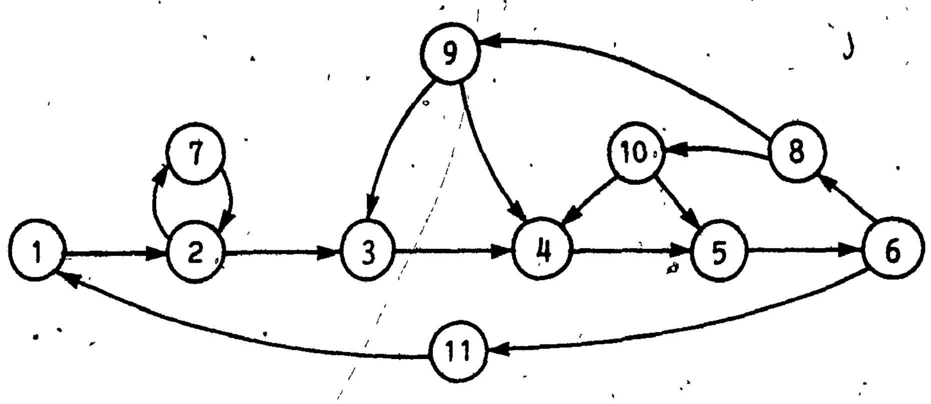
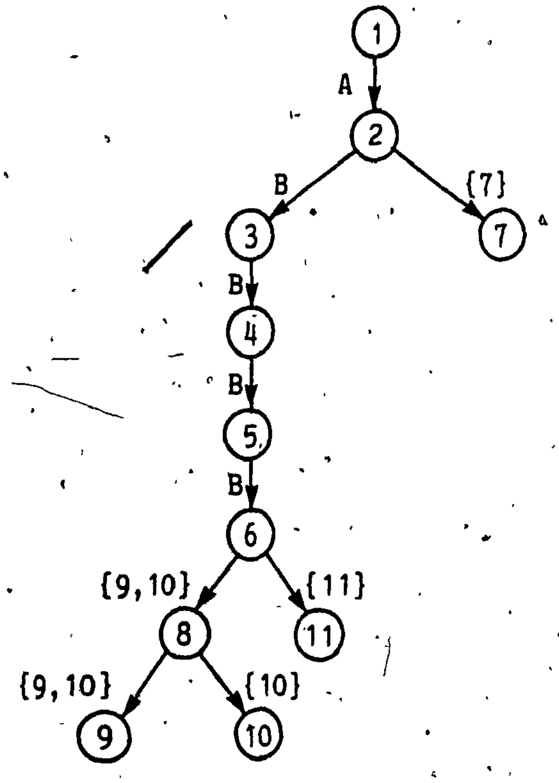


Figure 5.5a: Process graph 3



Edges are labelled with leaf Id. which are reachable from the root via that edge.

Legend:

A = { 7,9,10,11}

B = { 9,10,11}

Leaf Nodes:

T = {11}

N = {7,9,10}

Figure 5.5b: End of Phase-I

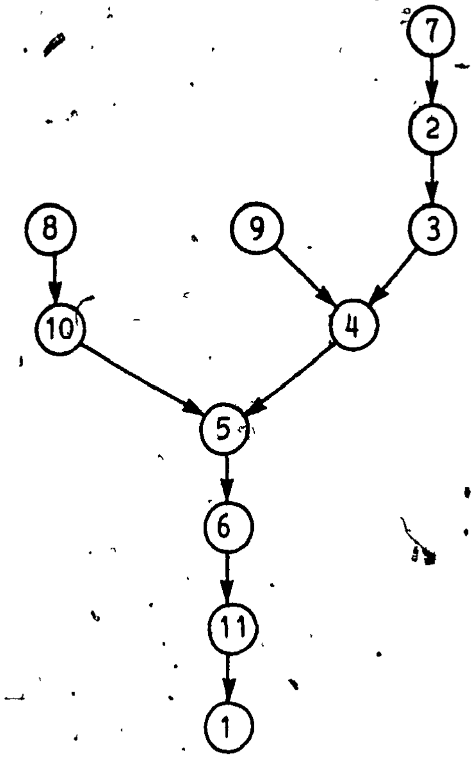


Figure 5.5c: Step 2 BFT$_r$

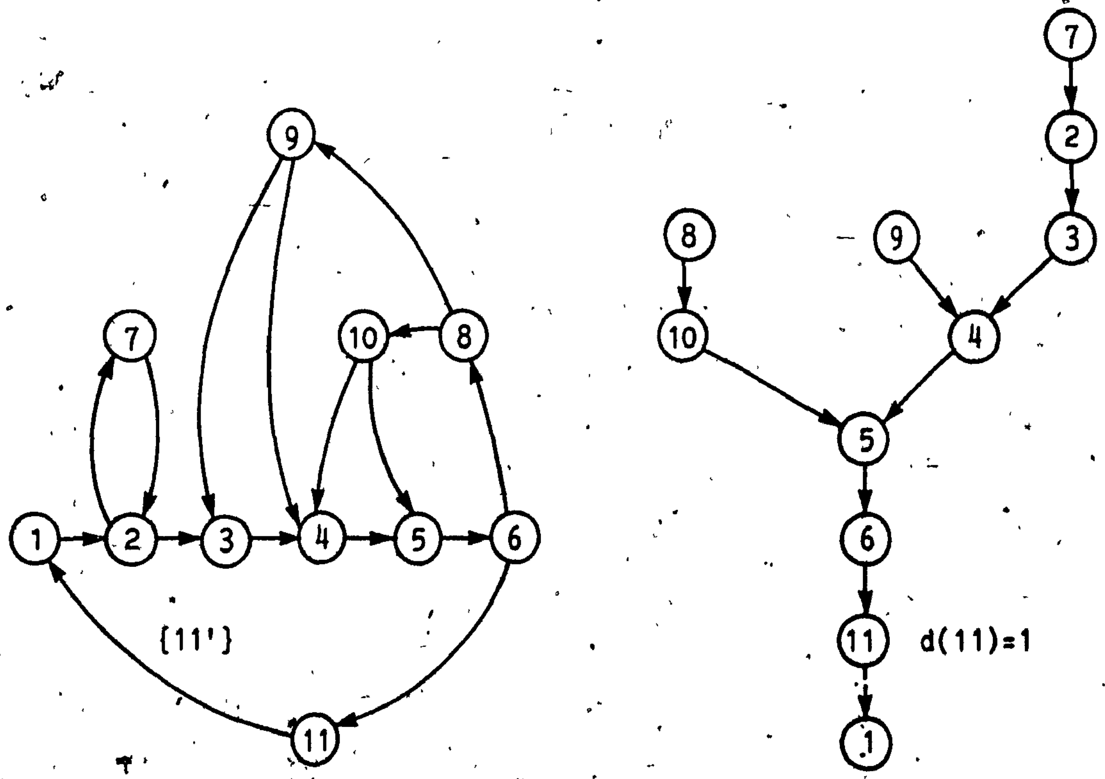


Figure 5.5d: Step 3 Labelling paths from nodes $\in$ T

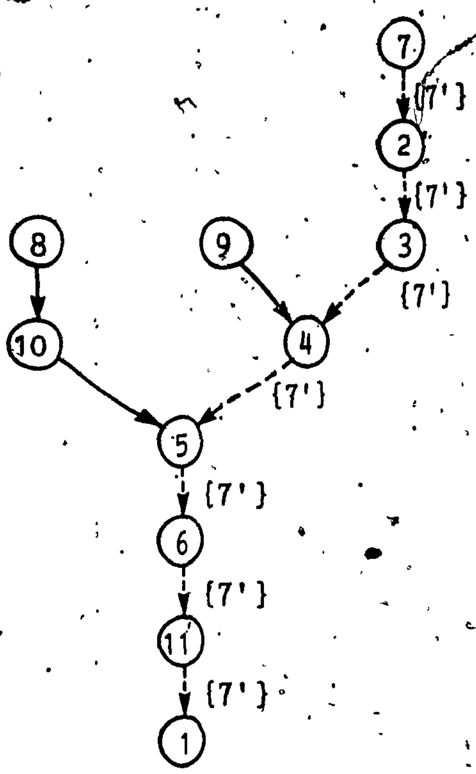
edges on the path from x through y to the coordinator R in the original BFT_r by x′. The corresponding edges in G are also labelled by x′: If either a second bucket exists, or x is found in the first bucket of the edge in G, then x′ and all y′ found in the first bucket are moved from the ~~first~~ bucket and placed in the second bucket; else x′ is placed in the first bucket. Remove these edges from the current BFT_r and repeat step 4 until $N = \emptyset$. The iterations are exemplified in fig.5.5e and the final labels of the chosen edges on which test messages are routed are shown in fig.5.5f. Table.5.1 tabulates the labels placed in the buckets of each of the edges in G.

Step 5: The Programming of a process node for forwarding test messages follows the rule:

At a node u, a test message is sent on an output edge corresponding to a bucket labelled {a,b,...} provided the test messages from its input edges corresponding to buckets that include labels {a,b,...} have been received, except that when a test message labelled with u is received (.i.e. $u \in T \cup N$), the node transforms the label to u′ before determining if a test message is to be sent out. During an iteration of clock update, corresponding to each non empty bucket a test message will be sent out only

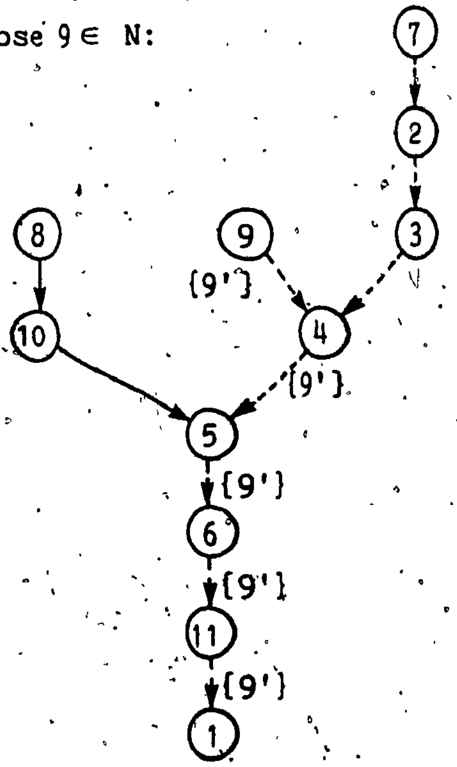
Step 4: N = {7,9,10}

Choose 7 ∈ N:



Example Complete Label in G: {9,10,11} ∪ {7'} = {7',9,10,11}

Choose 9 ∈ N:



Choose 10 ∈ N:

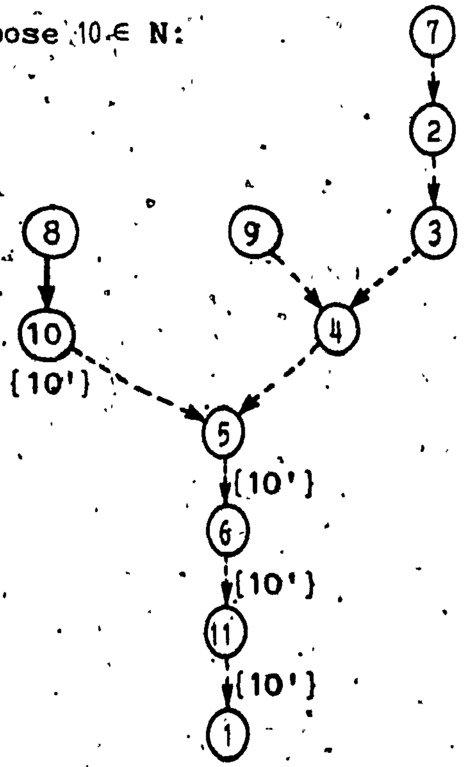
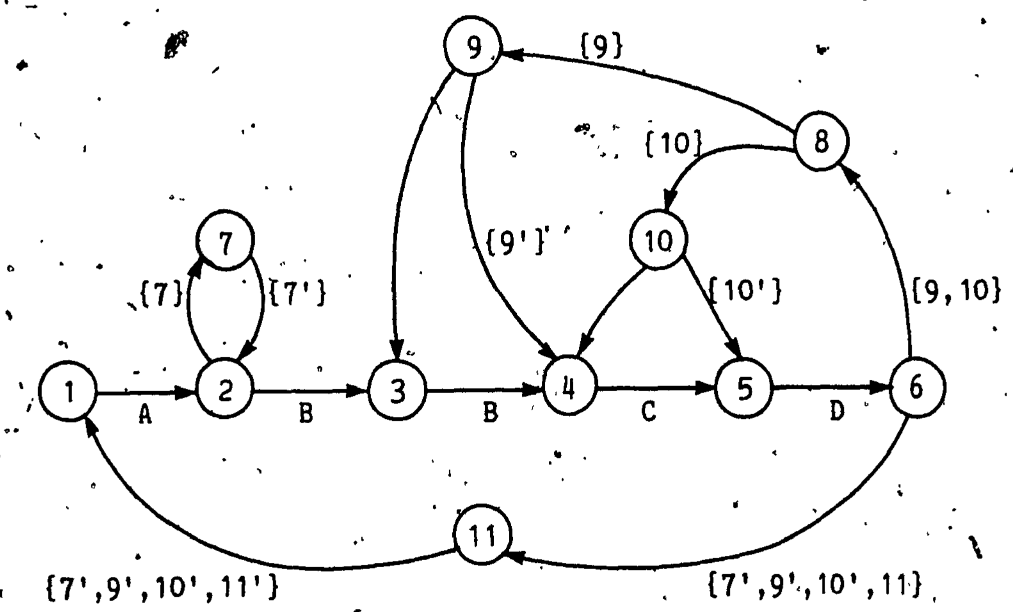


Figure 5.5e: Step 4 Labelling of reverse paths.



Legend:

A = {7,9,10,11}; B = {7',9,10,11}; C = {9,10,11},{7',9'};
D = {9,10,11},{7',9',10'}

Figure 5.5f: End of Phase II

Illustration of the labelling scheme

Edges	Bucket 1	Bucket 2
1,2	{7,9,10,11}	-
2,7	{7}	-
7,2	{7'}	-
2,3	{7',9,10,11}	-
3,4	{7',9,10,11}	-
4,5	{9,10,11}	{7',9'}
5,6	{9,10,11}	{7',9',10'}
6,8	{9,10}	-
8,9	{9}	-
8,10	{10}	-
9,4	{9'}	-
10,5	{10'}	-
6,11	{7',9',10',11}	-
11,1	{7',9',10',11'}	-

Table 5.1: Labels on edges.

once. For all edges in G containing empty buckets, a marker message is generated from the emanating process to flush the edge when the emanating process receives its first test message of this iteration. Marker messages are never forwarded.

Theorem 5.3

Two buckets are sufficient to hold all the labels that may be assigned to an edge in MINMES so that a label pair {x,x′} is never placed in the same bucket.

Proof: This is obvious from the construction as x′ is placed in the second bucket iff x is placed in the first or a second bucket already exists. Since the second bucket is created only during the consideration of the reverse paths x cannot exist in the second bucket and so x′ can definitely be put into the second bucket.

Q.E.D.

The implication of the above theorem is that an edge may have to transmit two test messages in the worst case. Algorithm MINMES is obviously an attractive heuristic as it minimizes the longest closed walk (therefore delay) from the coordinator R to the leaves and back to R, and at the same time it attempts to minimize repeated test message transmission.

The complexity of heuristic MINMES is $O(n^2)$, where n = number of processes, which corresponds to the iterative labelling applied in step 4.

Observe that MINMES is equally applicable as a heuristic for the modified subproblem for meeting Obj2 where the update iteration is supposed to involve only test messages. The heuristic possesses identical characteristic when applied to optimize this objective.

The MINMES algorithm for clock update can be easily extended to support non-FIFO channels [48]. Relaxation of the FIFO channel constraint implies that the control and event messages sent on the same channel could overtake one another on the channel. Hence the receipt of a control message (test or marker) will not indicate the flushing of the channel with respect to the previous iteration. So in order to ensure a process knows that it has completed the i^{th} simulation iteration and can respond to the $(i+1)^{th}$ test message for clock update, a test or marker message which may have reached a process ahead of an earlier event messages should carry with it the corresponding event list sent on that channel generated in the $(i-1)^{th}$ iteration. This redundancy is required to allow for non-FIFO channel behavior. Although the marker and test messages are longer as they contain the event lists, the performance will not be altered appreciably since in most communication subsystems the network throughput depends on the number of messages and not their lengths. Therefore performance characteristics of this extended version is similar to that of MINMES.

5.9 Performance Comparison of Bryant's Heuristic with MINMES

(a) Upper Bound of Message Overhead

Consider a process graph with O(n) nodes using Bryant's heuristic for clock update. Each process has O(n) input edges and forwards a test message upon receiving one from an input edge. Each test message sent out may percolate through O(n) edges to reach the coordinator. So the total number of probe messages is $O(n*n*n) = O(n^3)$. The flushing of edges takes an additional O(L) messages, where L = number of edges, so that an upper bound of the message overhead in Bryant's case is $O(L+n^3)$. This upper bound is tight and is required by the process graph shown in fig.5.6. In that case, the graph has a completely connected subgraph K_n. Process A receives n^2 probe messages originating from the various nodes. Returning these messages to process 1, the coordinator, requires n edge traversals each, leading to the complexity $O(n^3)$.

The algorithm MINMES by coordinating the forwarding of test messages will require (2n-2-k) edge traversals by test messages, where k is the number of edges which are shared by a forward and a return path. Since, each shared path will carry at most two test messages even in the worst case, the test message count is O(n). If the marker messages are included, the upper bound of message overhead becomes $O(L+k) \leq O(L+n)$.

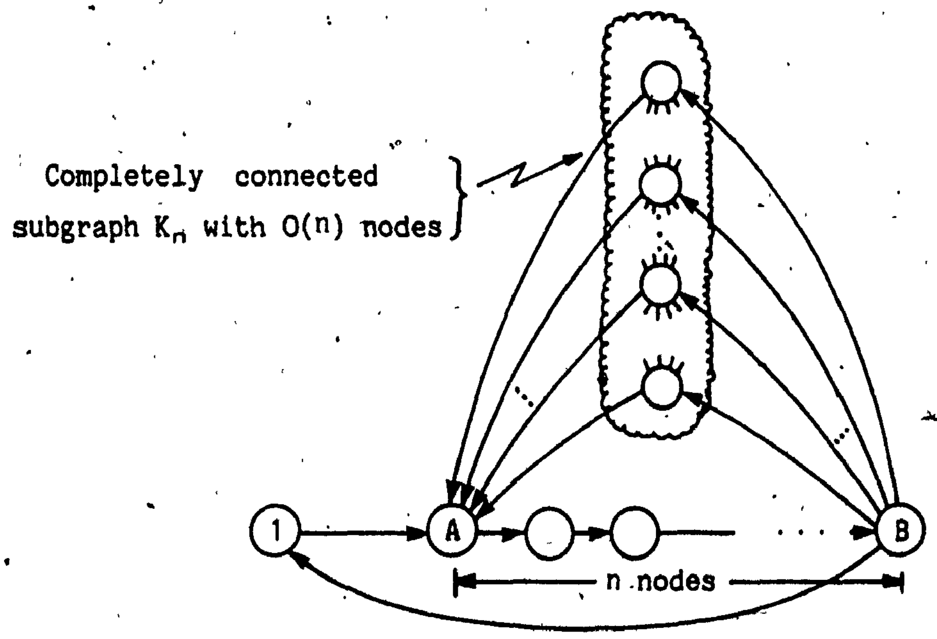


Figure 5.6: Strictness of Upperbound for Bryant's heuristic

(b) Lower Bound of Message Overhead

The lower bound in both heuristics are identical, and is equal to O(L). An example case is a cycle of n processes.

(c) Speed-up of MINMES

If we try to measure the speed of an update iteration by the longest path delay of message propagation from the coordinator to any node and back to itself where path delay is the number of edge traversals by a test message, we can immediately deduce that Bryant's algorithm has a worst case delay of $O(n^2)$ corresponding to the case of a test message routing through the n^2 edges in the K_n subgraph of fig.5.6 and then through A to B and back to the coordinator. MINMES on the other hand has a worst case delay of O(n) as evidenced by the traversal of BFT_f and BFT_r so that the height of either tree is at most O(n).

5.10 Simulation Concurrent with Clock update

In the explanation of the schemes given above, it has been assumed that the simulation is frozen while the clock update is in progress. However, instead of freezing the simulation completely while the clock update is in progress, a logically asynchronous clock update algorithm (eg. Time Incrementation [10]) can be executed in the background to improve concurrency. The coordinator can choose the opportune time for starting the next clock update iteration as follows. In the i^{th} update iteration in addition to determining the earliest invocation time (t_{min}) the second earliest invocation time (t_s) can also be estimated. If $t_s \geq t_{min}$ + Threshold, then the test messages corresponding to the $(i+1)^{th}$ iteration can be transmitted as soon as the coordinator becomes ready to perform the $(i+1)^{th}$ iteration. Otherwise, as the distributed update algorithm is more efficient for time increments $\leq$ Threshold, the coordinator should delay the commencement of the $(i+1)^{th}$ iteration until the distributed update algorithm has advanced the clock beyond t_s. Notice that t_s is only an estimate and the coordinator should not directly set the clocks to t_s because an event with invocation time $t < t_s$ may have been generated during the i^{th} simulation iteration. Such a hybrid heuristic reduces the overhead as it chooses the best strategy to be followed. The algorithm should be tuned for optimal performance by properly choosing the value of Threshold for the particular target application. It should be observed that, as the determination of t_s occurs along with the determination of t_{min} at each node and is carried in an additional field of the same test message, there is no appreciable performance penalty incurred by this scheme.

5.11 Summary

Efficient schemes for updating the simulation clocks in the processes modeled by strongly connected components of a process graph which implements discrete event simulation have been presented. Logically asynchronous solutions to this problem are inefficient and hence logically synchronous solutions have been developed. In these schemes, in order to perform a clock update, the global state of the distributed processes is compiled by a chosen coordinator, and the earliest invocation time of any event which is either in the event list of any process or in transit on some channel is determined. Unlike existing solutions, we keep track of the events that each process has created and transmitted to its neighboring processes, and compile the Global state simply by visiting all processes rather than flushing all edges.

This problem of detecting and compiling the global state is formulated as a directed graph traversal problem. A team of red and green tokens, starting from the coordinator node, travelling independently or in groups is required to traverse the graph such that (i) every node is visited by at least one red token, (ii) every edge is traversed by either a red or a green token, and (iii) all red tokens return to the coordinator. Solutions to this traversal problem have been developed so as to satisfy three different objectives of practical importance, namely, (1) Minimize the number of edges traversed by red and green tokens; (2) Minimize the number of edges traversed by red tokens; (3) Minimize the length of the longest walk from the coordinator to itself. An optimal solution has been presented for the third objective. Optimal solutions for satisfying objective 1 should coordinate and minimize the propagation of red and green tokens. We have shown that this is an NP-Complete problem. Consequently an efficient heuristic which can be used to achieve good solutions for both objectives 1 and 2 have been suggested. The heuristic involves a one time analysis of the process graph and its complexity is $O(n^2)$, where n = number of processes. The solutions for objectives 1 and 2, based on this heuristics have upperbounds on message overheads of $O(L)$ and $O(n)$ respectively, where L= number of edges. This compares very favorably with the existing solution whose upperbound is $O(n^3)$.

The modifications required for updating the clock in systems employing non-FIFO communication channels has also been suggested. The logically synchronous schemes assume that the simulation is frozen when the update increment is being assessed. However, in order to improve concurrency a hybrid scheme which combines a logically synchronous update algorithm with a logically asynchronous algorithm has been suggested. The performance of this algorithm must be tuned to appropriately initiate update using the logically synchronous scheme only when its estimate of the possible increment is greater than a threshold which is application dependent.

Chapter 6

Backward Error Recovery

Distributed Computer Systems are becoming increasingly popular and are being employed for critical applications demanding high reliability such as aircraft control, industrial process control and banking systems. Inherently DCSs are more complex than centralized systems. The added complexity could increase the potential for system faults. But, by introducing redundancy they do offer greater opportunity to realize fault tolerant systems and consequently in providing better system availability. A high proportion of failures in computer systems have been found to be due to residual design faults (hardware & software). They are the hardest to deal with as they are mostly unanticipated [39]. Usually these errors are handled by discarding the current system state, restoring the system to a prior state accepted to be correct and starting re-execution from there (Backward Error Recovery [2]). This requires the state of the system to be saved at different points (recovery points) during the computation. The procedure of saving the complete system state at a recovery point is known as checkpointing [2]. Upon detection of a failure, appropriate diagnosis and reconfiguration will be carried out. Subsequently, the system will be rolled back to one of the recovery points which has not been contaminated by the fault.

In distributed systems, an error occurring in one process might propagate to other processes. Hence a process can no longer be viewed in isolation. The rollback of a process might force the rollback of some other processes. If checkpointing of the different processes are unrelated to each other, then rollback of a process might result in an avalanche rollback of the entire system known as domino effect. In a DCS a set of component process states which together form a consistent state called a Recovery line can be identified and the overall system can be rolled back to this recovery line. Two strategies, "preplanned" and "unplanned" recovery, have been used in the literature to establish recovery lines [54]. In the preplanned case the checkpoints are synchronized at the time of state recording so that some objective recovery lines are formed [38]. In the unplanned strategy the system tries to deduce a recovery line when a rollback is imminent, thus requiring additional work and delay during rollback. Various schemes have been proposed for both strategies with some of them restricting the interprocess communication patterns [7,23,38,54].

Here we analyze the requirements for avoiding domino effects and then propose an efficient domino free rollback technique. The checkpoints established by the interacting asynchronous processes are coordinated dynamically and process-interdependencies are tracked so as to ensure minimal distance rollback, whenever rollback is necessary. Moreover this scheme neither requires the processes to be deterministic nor imposes restrictions on interprocess communication. Though the aspects of Error detection, Error containment and Error diagnosis [2] are very important for a fault tolerant system, they are orthogonal to the subject of this study and are not dealt with here. We assume that the state of every component process in the system can be rolled back as no limits on the error latency are assumed.

6.1 Fault tolerant system model

Every application process is appended with a control process. The control process has the responsibility of providing the checkpointing and rollback functions for its associated application process. The term process henceforth refers to the combination of an application process with a control process.

Application events of interest for this study are the Send, Receive and intraprocess events. Each message transmitted by a Send event u will be received by an explicit matching Receive event m(u). Similarly for every Receive event u, there is a matching Send event m(u).

The system events of interest are Create_self_induced_checkpoint (Csic), Create_response_checkpoint(Crc), Rollback, Send_rollback_message and Receive_rollback_message. In addition to this Fault events can occur in any of the processes. It is assumed that every fault will be diagnosed reliably within a finite period of time. In order to recover the system from the erroneous state, the process in which the error is detected will be forced to execute a rollback event which will retrieve a known good state (one closest to the fault) by reloading the process state saved by an appropriate checkpoint event. All application events executed in between this checkpoint event and the rollback event are suspect, and all effects of these events in the system must be undone. The distributed computation will be represented by the STM and the Pomset models.

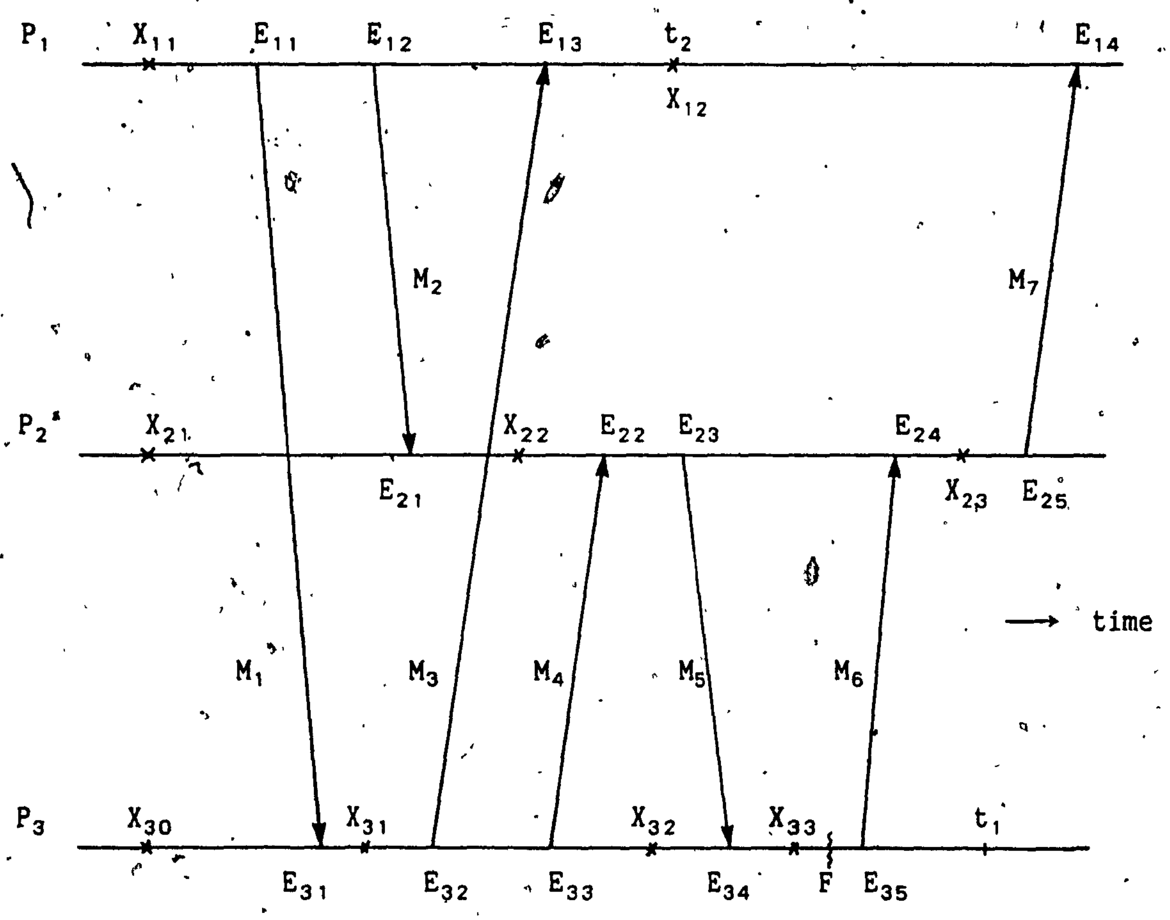
An instance of the computation of the system can be described by the Pomset $[V,\sum,\preceq,\mu]$. Events in each process are assumed to be totally ordered, whereas events across processes are partially ordered. The symbols have their usual significance.

6.2 Concepts of Rollback and Recovery

Consider the execution history of a process which is part of a distributed computation (say P_3 in fig.6.1). Assume that at time t_1 an error was detected in P_3 and is diagnosed to have occurred because of the faulty event "F". The following terms can be defined:

Definition: Error dependent set (EDS)

This set EDS(F) contains the earliest event (if any) in every process which may depend on the contaminated data produced as a result of the erroneous event F. During recovery for achieving minimal rollback the checkpoints just preceding the events in EDS(F) must be chosen for each of the dependent processes. In other words,



Legend

E : Event- Send or Receive X : Check-point

M : Message F : Fault

Horizontal Edges: Other Spontaneous Events

Cross Edges: Message Propagation

Figure 6.1: Distributed Computation - STM model

$$\mathrm{EDS}(F) = \{m(u) / (\mu(u) = \mathrm{Send}) \wedge (F \preceq u) \wedge$$
$$(\nexists v \text{ colocated with } m(u) \text{ s.t. } ((v \prec m(u)) \wedge (F \preceq v)))\}$$

Definition: Backward dependent set (BDS)

Assume that checkpoints X_{ia} and X_{jb} have been chosen for rollback in processes P_i and P_j respectively, where X_{ry} is the y^{th} checkpoint in P_r. The BDS(X_{ia}, X_{jb}) is defined as the set of all those Send events in P_j which precede X_{jb} and whose corresponding messages are accepted by Receive events in P_i which succeed X_{ia}. In other words, let Z_{ia} and Z_{jb} be the recording events of checkpoints X_{ia} and X_{jb} respectively. Then

$$\mathrm{BDS}(X_{ia}, X_{jb}) = \{u / (\mu(u) = \mathrm{Send}) \wedge (u \text{ is colocated with } Z_{jb}) \wedge$$
$$(m(u) \text{ is colocated with } Z_{ia}) \wedge (u \preceq Z_{jb}) \wedge$$
$$(Z_{ia} \not\preceq m(u))\}$$

Definition: Retransmission dependent set (RDS)

Assume X_{ia} and X_{jb} in P_i and P_j as before. The RDS(X_{ia}, X_{jb}) is defined as the set of all receive events in P_j preceding X_{jb} which accept messages sent by send events in P_i executed after X_{ia}. In other words, let Z_{ia} and Z_{jb} be corresponding recording events as before. Then

$$\mathrm{RDS}(X_{ia}, X_{jb}) = \{u / (\mu(u) = \mathrm{Receive}) \wedge (u \text{ is colocated with } Z_{jb}) \wedge$$
$$(m(u) \text{ is colocated with } Z_{ia}) \wedge (u \preceq Z_{jb}) \wedge$$
$$(Z_{ia} \preceq m(u))\}$$

The events in EDS(F) and all their successors in each of the processes can be thought of as defining a Fallout region for the fault F as illustrated in fig.6.2a. During recovery the Recovery line which just envelopes the Fallout region corresponding to the diagnosed fault must be chosen in order to ensure minimal rollback. Normally the Send events in BDS(X_{ia}, X_{jb}) of a pair of checkpoints belonging to the chosen Recovery line, will have to be re-executed in order to resend the messages and satisfy the receive requirements during re-execution following rollback. The cross edges representing these messages, in the event graph of the computation, will intersect every cut containing the pair of checkpoints X_{ia} and X_{jb}. These cross edges will be Forward edges with respect to these cuts as illustrated in fig.6.2b. The Receive events in RDS(X_{ia}, X_{jb}) of a pair of checkpoints belonging to the chosen Recovery line, will have to be re-executed in order to consume the new messages that are generated because of the re-execution of the corresponding Send events by the Sender process. The cross edges representing these messages, in the event graph of the computation, will intersect every cut containing the pair of checkpoints X_{ia} and X_{jb}. These cross edges will be Backward edges with respect to these cuts as illustrated in fig.6.2c.

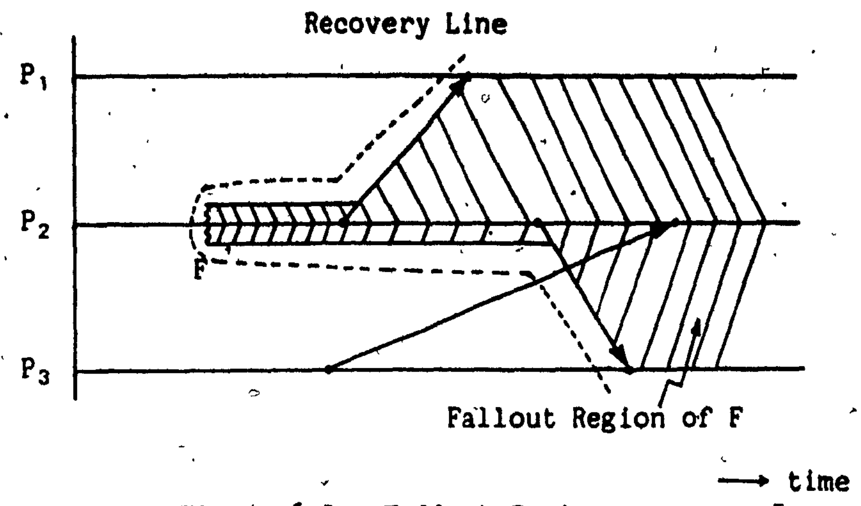


Figure 6.2a: Fallout Region

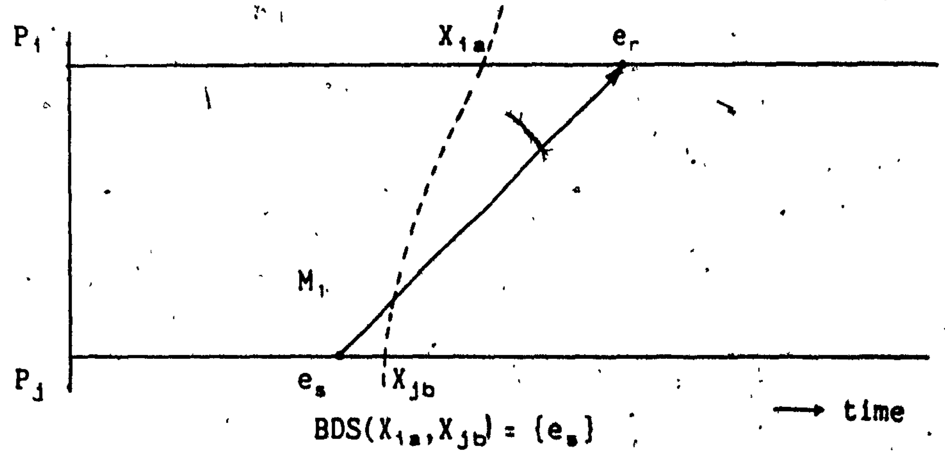


Figure 6.2b: Forward Edge

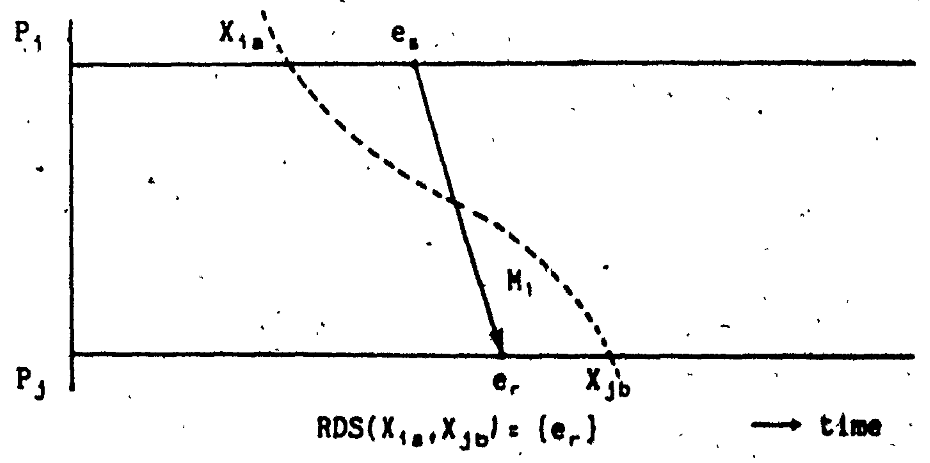


Figure 6.2c: Backward Edge

6.2.1 Domino rollback in distributed systems

Asynchronously establishing checkpoints in the processes of the system and then requiring the system to rollback only to recovery lines whose corresponding cuts do not intersect any cross edges (Backward or Forward), may lead to domino rollback. This is illustrated by considering the example of fig.6.1. At t_1 let an error be detected in P_3. Let the diagnosis indicate event-F as the cause. So P_3 should be rolled back ,to checkpoint X_{33}. As EDS(F) contains $\{E_{14},E_{24}\}$, P_1 and P_2 have to be rolled back to X_{12} and X_{22} respectively in order to overcome the effects of errors. However, $\{X_{12},X_{22},X_{33}\}$ does not form a legitimate recovery line because BDS(X_{22},X_{33}) contains E_{33} and RDS(X_{22},X_{33}) contains E_{34} and both are non empty. Observe that if either RDS or BDS is non empty it implies that there are cross edges intersecting the corresponding recovery line. This will induce further rollback in the processes until the legitimate recovery line $\{X_{11},X_{21},X_{30}\}$ is reached. In other words, checkpoints (state recording) are established in processes asynchronously and later an effort is made to determine a Synchronized cut (Synchronized prefix) using the checkpoints, to which the system can be rolled back. As explained in §4.5 such a synchronized cut is not guaranteed to be constructable for all distributed computations. Moreover for a certain sequence of computation, the recovery lines might be few and far apart leading to extensive rollback during recovery and sometimes even unbounded rollback (domino-effect).

A simple solution is to adopt the Conversation scheme [§4.5,38] which is a planned strategy for checkpointing. In this scheme the interprocess communication patterns are restricted to enable Synchronized cuts to be constructed during the execution by synchronizing the processes. Such a scheme is quite inefficient due to the periodic synchronization that has to be established. Therefore it is necessary to examine strategies for eliminating RDS and BDS in asynchronous checkpointing schemes.

Domino rollback might also occur when processes are allowed to suspect the messages it receives from others [23]. On encountering an error, processes might end up blaming each other as the cause of the error and might thus trigger a chain of rollbacks. Normally it is assumed that the received messages are always correct. However this will require that each process be capable of diagnosing all its errors. Additionally although it might be able to diagnose errors of other processes easily it is prevented from doing so. In the interest of faster recovery at least limited handling of these suspicious messages must be supported by the rollback scheme. The following can be asserted using the formalism given above.

Theorem 6.1

For any fault F, minimal Domino-free rollback is guaranteed if from the Recovery line composed of the checkpoints immediately preceding the events in EDS(F), the Backward and Retransmission dependencies are eliminated

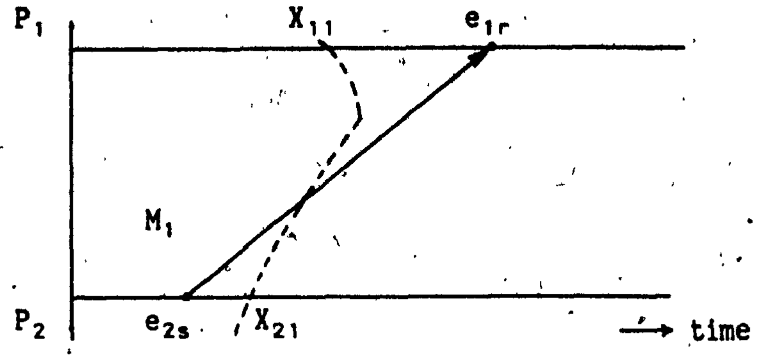
Proof

Part (a): Domino Free

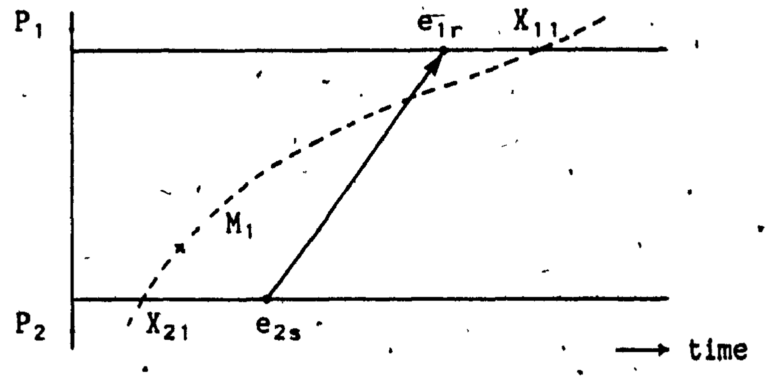
This will be proved by contradiction. If possible assume that domino effect occurs even if backward and retransmission dependencies are eliminated from the recovery line composed of the checkpoints immediately preceding events in EDS(F). Let P_1 and P_2 be two processes and let F be a fault. Let the corresponding Recovery line be composed of checkpoints $\{X_{11}, X_{21}\}$ immediately preceding EDS(F). If rollback to this is to produce a domino effect, then the message dependencies of either of the two processes is not satisfied (fig.6.3) in rolling back to X_{11} and X_{21}, i.e. either $RDS(X_{11},X_{21})$ or $RDS(X_{21},X_{11})$ or $BDS(X_{11},X_{21})$ or $BDS(X_{21},X_{11}) \neq \emptyset$. This is an obvious contradiction.

Part (b): Minimal rollback

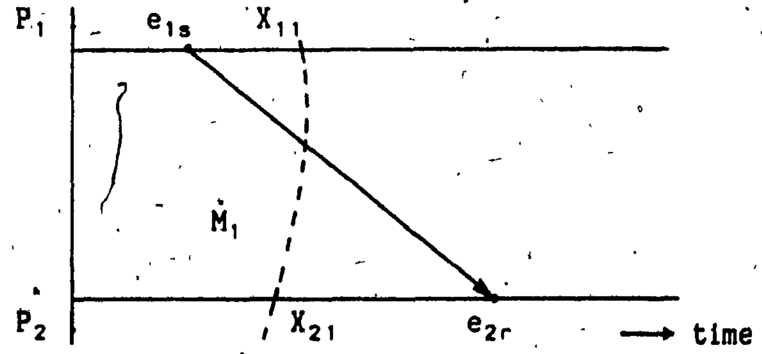
The recovery line chosen is composed of the checkpoints immediately preceding EDS(F). This obviously is the recovery line which will just encompass the fallout region. Hence no other recovery line which guarantees correctness and yet decreases the rollback in the processes



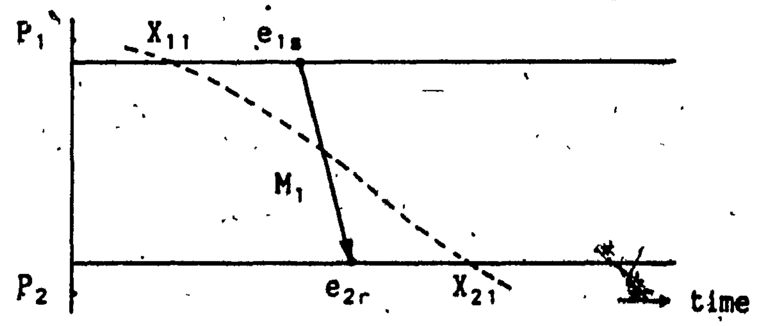
$\mathrm{BDS}(X_{11},X_{21}) = \{e_{2s}\}$



$\mathrm{RDS}(X_{21},X_{11}) = \{e_{1r}\}$



$\mathrm{BDS}(X_{21},X_{11}) = \{e_{1s}\}$



$\mathrm{RDS}(X_{11},X_{21}) = \{e_{2r}\}$

Figure 6.3: Non Empty BDS and RDS

exists.

Q.E.D.

Based on the above theorem the following can be defined.

Definition: Valid Recovery Line (RL)

A set of checkpoints $(X_{1a}X_{2b}...X_{is}X_{jr}...)$ constitute a Valid recovery line RL, with respect to a fault F in a process P_d provided:

(a) $\forall j$: Z_{jr} which records X_{jr} precedes E_j $\forall$ $E_j \in$ EDS(F).

(b) $\forall j$: If $X_{jr} \in$ RL then $X_{jm} \notin$ RL for any $m \neq r$

(c) $\forall i,j$: RDS$(X_{is},X_{jr}) = \emptyset$

(d) $\forall i,j$: BDS$(X_{is},X_{jr}) = \emptyset$

This implies that none of the events belonging to a Valid recovery line should depend on one another.

6.2.2 Strategies for elimination of Backward dependencies

Backward dependencies occur because of undoing Receive events during rollback without undoing the corresponding Send events. In the following strategies to eliminate these backward dependencies are presented:

Strategy A: All received messages are saved during execution. During re-execution following rollback the messages needed for satisfying the receive events corresponding to the BDS are selectively retrieved from the repository.

Strategy B: Only those messages required to satisfy the BDS requirements are saved during execution. In the STM of fig.6.1 consider a line joining checkpoints X_{22} and X_{32} of two processes P_2 and P_3 respectively. The send events corresponding to these messages which are to be saved occur before the line $X_{22}X_{32}$ and the receive events occur after the line. This reveals that the messages to be saved are those which intersect the $X_{22}X_{32}$ line. In general, only those messages whose corresponding edges in the event graph are Forward edges w.r.t the Recovery line need be saved in order to satisfy the Backward dependencies during re-execution.

6.2.3 Strategies for elimination of Retransmission dependencies.

Retransmission dependencies occur because of undoing Send events during rollback without undoing the corresponding Receive events.

Strategy C: Observe that the Send events corresponding to the Receive events of RDS, precede the events in EDS(F). Therefore these messages are not erroneous. If the computation is assumed to be repeatable (functional system), these Send events will definitely be re-executed as they precede events in EDS(F) and the messages generated by these events during re-execution will be the same as the ones sent earlier. Hence, the new message can be regarded as redundant and ignored. Thus this scheme will not require the receiver to be rolled back in order to satisfy the RDS requirements.

Strategy D: Coordinate setting up of checkpoints of processes so that recovery lines do not contain RDS. In other words the checkpoints of the processes are recorded such that there are no Backward edges w.r.t the recovery line defined by these checkpoints.

In a general distributed system it is impractical to assume exact repeatability of events [52], on account of the nondeterminism inherent in the system, variable message delays, nondeterministic behavior of certain high level language statements like the Select statement in ADA, and presence of real time events. Therefore the scheme proposed in this thesis will use Strategy D in order to eliminate the Retransmission dependencies. The proposed scheme minimizes storage overhead by adopting Strategy B in order to eliminate the Backward dependencies. The coordination required in establishing checkpoints for eliminating RDS is carried out without freezing the computations.

6.3 Dependency based Checkpointing and Recovery

Recovery lines constructed using strategies B and D for eliminating BDS and RDS respectively, define Consistent cuts in the event graph of the computation. The resulting Consistent global state is used for the purposes of rollback. The recoverability property of Consistent global states ensures the correctness of such a rollback recovery scheme. However, we observe that all processes may not get corrupted by a faulty process. Therefore in the interest of minimal rollback, rollback must be restricted only to those processes which have received messages either directly or indirectly from the faulty process, sent after the occurrence of the fault. In order to dynamically keep track of such interacting set of processes and precisely define the fallout regions the interdepencies between the processes will have to be identified. This identification scheme will henceforth be referred to as "dependency tracking".

The rollback recovery scheme presented here is based on dependency tracking and the Consistent global state detection techniques [29,11]. This, also minimizes unnecessary rollback. An application process may initiate local checkpoints, by executing Csic events, based on its local requirements. Subsequent to a local checkpointing, messages sent from this process to others may induce setting up of checkpoints in other processes, by making them execute Crc events, on receiving these messages. In terms of the global state detection scheme, every process which creates a self induced checkpoint is an initiator. Marker messages are eliminated in this case since dependency tracking is required. The required control information for dependency tracking, and inducing coordinated checkpoints is piggybacked on the application messages themselves. Additionally, a global state compilation phase is not necessary since the checkpoint information is assumed to be stored and utilized locally in each of the processes. Therefore coordinated checkpointing does not require any special control messages. However during rollback the processes will forward a special control message called the "rollback message". During recovery the affected processes will be rolled back and the contaminated messages will be discarded. This rollback recovery is achieved without freezing the ongoing application computation.

6.3.1 Checkpointing algorithm

A process P_i creates checkpoints based on its local requirements by executing Csic events. Each such checkpointing or state recording is associated with a unique identification tag. X^i_{iq} denotes the qth self induced checkpoint of process P_i. The structure of a checkpoint is shown in fig.6.4a. A n dimensional checkpoint vector denoted as CCP is associated with every process P_j, It contains the ordinal number of the latest self induced checkpoint created by each of the n processes P_i in the system, as perceived by P_j. The CCP vector is appended to every application message sent out by that process. A process P_j upon receiving an application message strips the checkpoint vector from the received message (call it RCP), compares RCP with the its CCP and updates the CCP appropriately. The operations performed by the receiver process based on the comparison of the two vectors is given in the form of pseudo code in fig.6.4b and illustrated as follows.

Let process P_j receive an application message M with vector RCP. In the RCP let r be the value corresponding to process P_k. If r is greater than the corresponding value in the local CCP, then P_j creates a response checkpoint X^j_{kr}, where X^j_{kr} represents that this checkpoint is that

Organization of Checkpoint information

Type	RMN	DMN	XRL	XCP	PS	MS

Type:	Self induced or Response Checkpoint.
RMN :	Value of the Receive message counter when the checkpoint was created.
DMN :	Receive Message counter value of the earliest message that should be discarded on rolling back to this checkpoint.
XRL:	Recovery line <Pid,No> to which this checkpoint belongs, where Pid: is the Process Id of the owner, and No: is the ordinal number of the Self induced checkpoint in the owner corresponding to this Recovery line.
XCP:	Reflects the Checkpoint vector of the process when this checkpoint was created.
PS :	The state of the process when this checkpoint is created.
MS :	Set of messages on the input channels of the process which intersect the Recovery line corresponding to this checkpoint.

Figure 6.4a: Structure of a checkpoint

```
Create Self induced checkpoint in process $P_i$
(1) Create new checkpoint of $P_i$:
        Type := Self induced;
        RMN := Receive Message counter (RecMsg); DMN := RMN + 1;
        XRL := <i,CCP(i)>; XCP := CCP;
        Record process state in PS (State prior to this event); MS := Ø;
(2) Update CCP(i)


Functions to be carried out by any process $P_j$ on receiving a message $M_x$
:
(1) Update RecMsg; Temp := CCP
(2) Strip the checkpoint vector RCP from $M_x$
(3) Examine corresponding entries of RCP and CCP
        and repeat steps 3a and 3b for i = 1 to N.

 (a)  If RCP(i) > CCP(i) then
          Temp(i) := RCP(i);
          Create new checkpoint of $P_j$: Type, RMN, DMN, XRL, XCP, PS, MS of
                  checkpoint to Response, RecMsg, RecMsg, <i,CCP(i)>, CCP,
                  Process state, and Ø respectively.
 (b)  If RCP(i) < CCP(i) then
          For all checkpoints of $P_j$ owned by $P_i$ which have No.XRL > RSP(i)
                  append <RecMsg,$M_x$> to MS of the checkpoint.
          If checkpoint <i,RCP(i)> does not exist then
                  Create Checkpoint by copying Type, DMN, PS and MS of
                  Checkpoint <i,k>, where k is the smallest integer > RCP(i);
                  RMN := RecMsg; XRL := <i,RCP(i)>;
                  Delete $M_x$ from MS of this checkpoint;
(4) CCP := Temp;
```

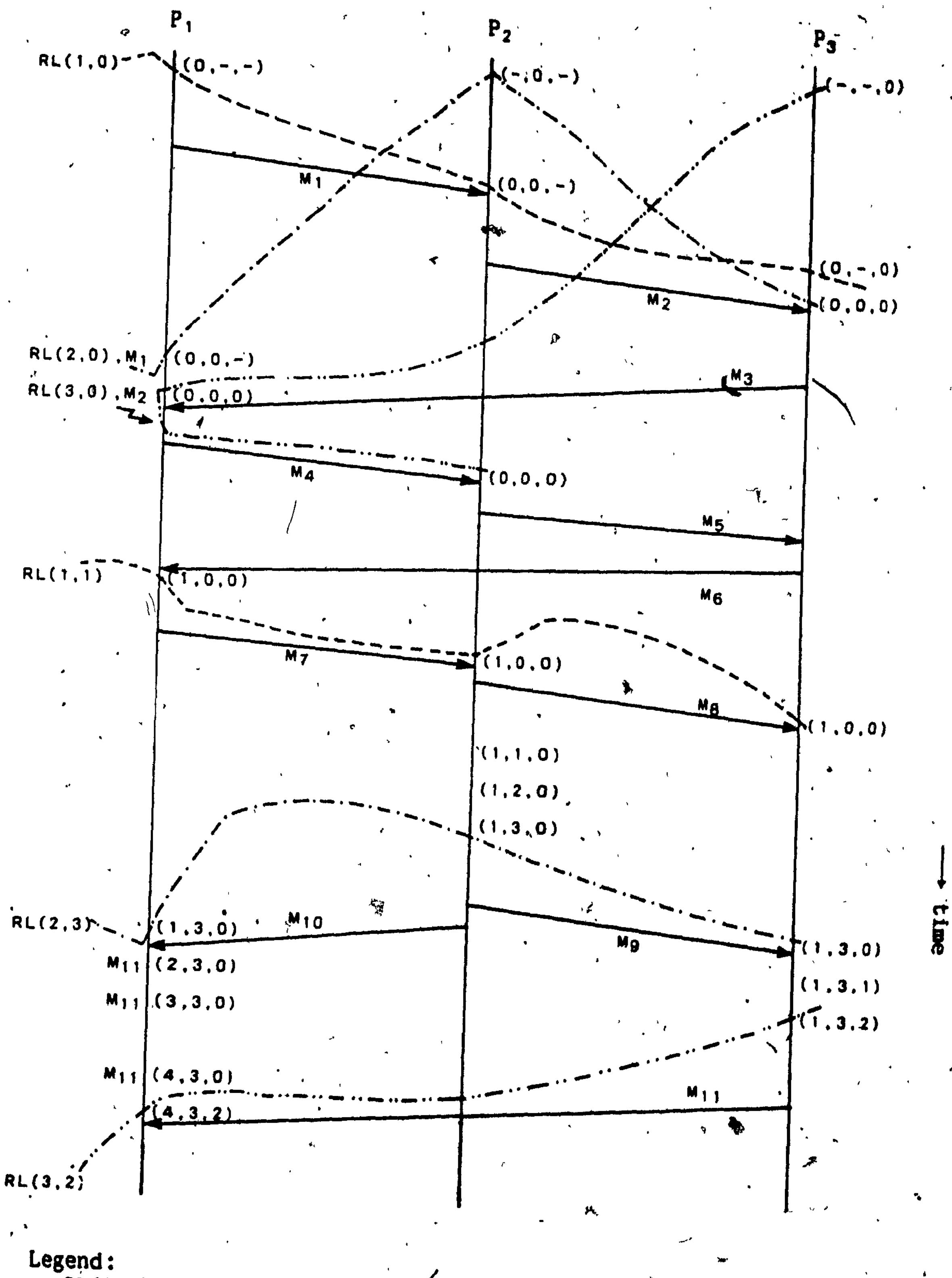
Figure 6.4b: Receive Procedure

of P_j and has been created in response to the r^{th} self induced checkpoint of P_k. The process state (PS) field of the checkpoint contains the current value of the process variables. The Message state (MS) field of the checkpoint is set to a null value. The Receive Message Number (RMN) is set to the current value of the Receive Message counter and the Discard Message Number (DMN) field is set to (RMN + 1). This checkpoint is now regarded as being owned by P_k and forms a part of the recovery line $\langle k,r \rangle$. However, if r is smaller than the corresponding value in CCP then P_j inserts this message (M) to the MS fields of all previous checkpoints X^j_{ky} of P_j satisfying the relation $y > r$. In addition, if P_j does not have the checkpoint X^j_{kr} then such a checkpoint is created. This is done by copying the information of checkpoint $X^j_{kr'}$ of P_j (where r' is the smallest integer greater than r for which a checkpoint of P_j owned by P_k exists) excluding message M. Exclusion of message M is necessary in order to ensure that it does not appear twice during the re-execution. In this case as before the RMN of X^j_{kr} will be the current Receive Message counter value, however the DMN will be the DMN value of X^j_{ky}. So though the checkpoint X^j_{kr} is created later, the information contained will reflect the state of the process just prior to the establishment of X^j_{ky}. This is necessary in order to strictly maintain the causality relationship. Such a checkpoint is known as a copy checkpoint. When y equals r, nothing needs to be done since the checkpoint is already established.

Thus a "cut" or recovery line is dynamically established and the message intersecting such recovery lines are identified. Each recovery line RL_{yz} is specified by the quadruple (y,z,W_{yz},B_{yz}). This implies that the recovery line RL_{yz} has been initiated by the z^{th} checkpoint of process P_y. W_{yz} is the ordered set of checkpoints in the processes $(P_1...P_n)$ which belong to this recovery line. B_{yz} is the set of messages which cross this recovery line. This set of checkpoints and messages are distributively saved throughout the system. Each process saves only the messages which correspond to its input channels. The recovery line is confined only to the group of processes subsequently interacting with the initiator of the recovery line either directly or indirectly, and related checkpoints are established only based on message flow. The number of recovery lines set up depends on the frequency of checkpointing in a process and the subsequent mutual interactions. Salient features of this checkpointing scheme are shown in fig.6.5. However, the derivation of checkpoints and relevant recovery lines are explained via an example in Appendix B.

6.3.2 Recovery of Processes

Upon detection of an error a process performs error diagnosis and identifies a recovery line say RL_{kq} to which the system must be rolled back in order to undo all computations performed after the system has entered the faulty state. The pseudo code for this rollback procedure is given in fig.6.6. The process is rolled back to the chosen recovery line by loading its process state variables with the values saved in the PS field of



Legend:

RL(X,Y),Z	: Recovery line numbered Y of Process X and Message-Z is attached to this Recovery line.
(a,b,c)	: Checkpoint Numbers of Process P_1,P_2 & P_3 respectively.
M_x	: Message numbered x.

Figure 6.5: Illustration of the Checkpointing Algorithm

Functions to be carried out during rollback recovery

Case (a): A process P_i either might detect an error, diagnose it and select a recovery line $<r,y>$ to which it should rollback to overcome the error, or the process might receive a recovery message $<r,y>$ on a channel not anticipating this rollback message (say input channel-qi).

(1) Retrieve and load process state of checkpoint X^i_{rz} corresponding to the Recovery line $<r,z>$, where z is the smallest integer $\geq$ y, for which a checkpoint in P_i belonging to a recovery line owned by P_r exists.

(2) Discard all checkpoints of P_i with RMN $\geq$ DMN of X^i_{rz}.

(3) Insert messages belonging to MS of X^i_{rz} at the head of the respective input channel buffers.

(4) From MS of all retained checkpoints of P_i discard messages which have ordinal numbers $\geq$ DMN of X^i_{rz}

(5) Discard all application messages from all input channel buffers which directly or indirectly depend on the checkpoint X^r_{ry} of P_r.

(6) If P_i is the initiator of the rollback then set all input channels (if P_i is not the initiator then set all input channels except $\text{channel}_{q,i}$) to the Cautious state and append R<r,y> to the ARM list of these channels.

(7) Send Rollback message R<r,y> on every outgoing channel.

Case (b): Process P_i receives the rollback message R<r,y> on a channel which is in the Cautious state and is anticipating this rollback message.

(1) Delete this entry from the ARM list of the channel. If due to this the ARM list of the channel becomes empty, then set the channel to the Normal mode.

Figure 6.6: Rollback Procedure (contd.)

Receipt of application messages

Application message M_x on channel$_{qi}$ is received by a low level control procedure which identifies and discards erroneous messages

(1) If channel$_{qi}$ is in Normal Mode then Append M_x to the channel buffer of channel$_{qi}$

(2) If channel$_{qi}$ ~~is in~~ Cautious mode, then discard message M_x if its RCP depends on any of the rollback messages in the ARM list of this channel.

Figure 6.6: Rollback Procedure

the corresponding checkpoint and inserting the messages saved in the MS field of this checkpoint to the head of the appropriate input channels. All checkpoints with RMN $\geq$ DMN of RL_{kq} are discarded and from all retained checkpoints messages whose RMN $\geq$ DMN of RL_{kq} are deleted. Then it sends out rollback messages $R\langle k,q\rangle$ on all its output channels. When another process P_j receives this rollback message for the first time, it rolls back to its checkpoint X^j_{kr} (where r is the smallest number $\geq$ q for which the checkpoint owned by P_k exists) by setting the process variables to the values saved in the PS field of the checkpoint, and inserting the messages saved in the MS field of this checkpoint to the heads of the appropriate input channels. Subsequently, this process sends the rollback message $R\langle k,q\rangle$ on all its output channels. The messages which depend on the rolled back states of P_k should be discarded as they are dependent on some faulty computation. In order to accomplish this, when a rollback message $R\langle k,q\rangle$ is received for the first time by a process it will put all its other input channels into a cautious state in anticipation of the same rollback message $R\langle k,q\rangle$ to be received on them. A list called the ARM (Anticipated Rollback Message) list is maintained for each input channel of every process. The entry $\langle k,q\rangle$ will be inserted into the appropriate ARM lists. A channel which is in the cautious state and has $\langle k,q\rangle$ as an element in its ARM list, will discard every message in the channel whose RCP value y for P_k is $\geq$ q. Eventually when the rollback message $R\langle k,q\rangle$ is received on a channel the corresponding entry $\langle k,q\rangle$ will be deleted from its ARM list. A channel is returned from the cautious state to the normal state when its ARM list becomes empty. Thus the erroneous messages are revoked by the sender in cooperation with the receiver process. The overhead involved during rollback is O(L) control messages where L is the number of channels in the system.

Consider the event graph shown in fig.6.5. Let process P_3 detect an error and decide to rollback to the recovery line RL_{30} i.e. the zeroth checkpoint of P_3. Then P_1 and P_2 will be rolled back to the corresponding checkpoints. All the channels will be empty except for the $channel_{23}$ which will be loaded with message M_2. Rolling back of processes P_1, P_2 and P_3 are not synchronized and hence their computations are not frozen during rollback.

Consequently, the effect of a rollback is to restore the system (affected processes) to a previously known good state. As the main aim of rolling back the system is to overcome all effects of the diagnosed fault on the system the action of the rollback event can be interpreted as substituting a Nop: No operation event, for the segment of the pomset representing the computation between the chosen checkpoint and the rollback event. Using this interpretation the strategy used in creating copy checkpoints can be justified. Suppose X^{i}_{kq} was a copy checkpoint created using information from X^{i}_{ky}. The intention of rollback is to anull the effect of all events in the system that are dependent on X^{k}_{kq}. Consequently, the effect of the message M which created the checkpoint

X^i_{kq} in P_i must be anulled as the send event of message M is dependent on X^k_{ky} (either directly or indirectly) which in turn is dependent on X^k_{kq}. Therefore P_j will have to be rolled back to the state it had just prior to the establishment of X^i_{ky}. Hence, the information for X^i_{kq} is copied from X^i_{ky} where y is the smallest integer > q for which a checkpoint of P_i belonging to a recovery line owned by P_k exists.

Theorem 6.2

The dependency based coordinated checkpointing and rollback recovery schême is domino effect free.

Proof

Let F be the fault diagnosed to be in process P_i. Let its self induced checkpoint which just precedes fault F be X^i_{ik}. Let RL_{ik} designate the recovery line corresponding to X^i_{ik} which is constructed by the Checkpointing algorithm. Now to prove that:

(1) RL_{ik} is indeed a Valid recovery line.

(2) Rollback is Domino-free.

Part 1: The proof is based on contradiction. If possible let RL_{ik} not be a Valid recovery line, i.e. one of the four properties of Valid recovery lines (§6.2.1) is not satisfied.

(i) Violation of Property a: No effect of Fault F can precede it. Hence obviously no effect can persist in P_i as the chosen recovery line RL_{ik} precedes this fault. However, if possible let an effect in process P_j due to fault F not be overcome by rolling the system back to RL_{ik}. Then P_j should have been influenced either directly or indirectly by P_i after fault F had occurred. Such an influence can only be through the receipt of a faulty message. Let M_z be such a message (fig.6.7a). Since it has been assumed that an effect of the fault F was not overcome by rolling back P_j to X^j_{ik}, definitely the receive event of M_z must precede the checkpoint X^j_{ik}. Hence message M_z will be represented by a Backward edge (T partition to S partition) in the corresponding event graph. This is a contradiction since the checkpointing scheme does not allow recovery lines to be intersected by Backward edges.

(ii) Violation of Property b: If possible let there be two checkpoints of P_j in RL_{ik}. If $i \neq j$ these checkpoints must be response checkpoints. Therefore these two response checkpoints must have been created either due to the receipt of one message, or due to the receipt of two different messages. From the code of the checkpointing algorithm it can be observed that a response checkpoint X^j_{yz} is created and associated with some recovery line RL_{yz}, only if this checkpoint had not been created before. Hence it can be immediately inferred that two response checkpoints of a process cannot be associated with the same recovery line. Similarly neither can two self induced checkpoints be associated with the same recovery line, since the ordinal number assigned to the checkpoint will be incremented immediately after establishing a self induced checkpoint.

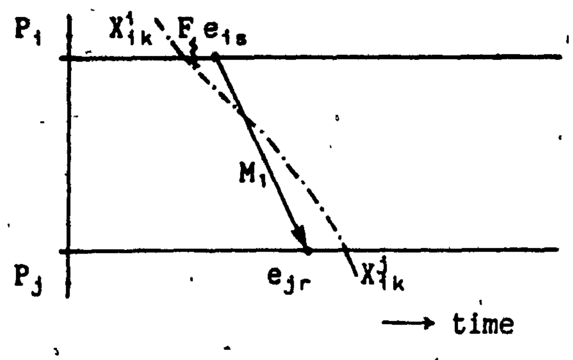


Figure 6.7a: Persisting effect of an event in EDS(F)

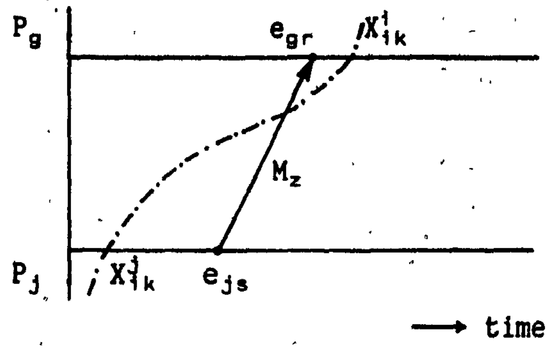


Figure 6.7b: Non Null RDS

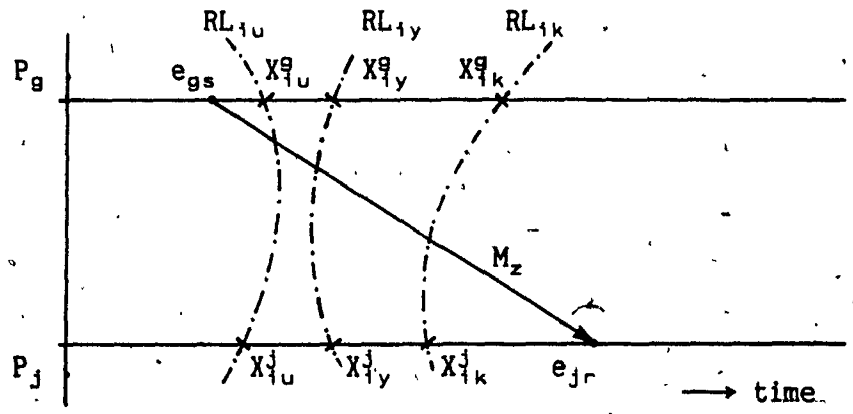


Figure 6.7c: Non Null BDS

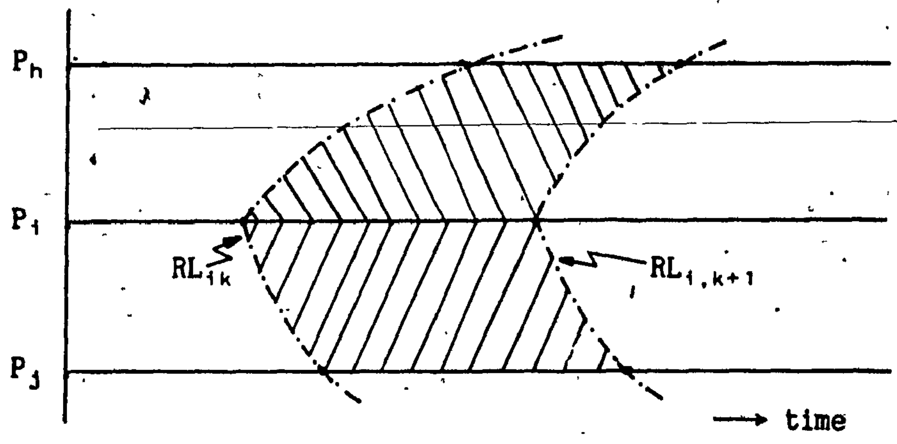


Figure 6.7d: Boundedness of Rollback

(iii) Violation of Property c: Let X_{ik}^{j} and X_{ik}^{g} be two checkpoints in RL_{ik}. If possible let $RDS(X_{ik}^{j},X_{ik}^{g})$ be non null. This implies that there is at least one message M_s, transmitted from P_j and accepted by P_g, whose Receive event in P_g precedes X_{ik}^{g} and whose Send event in P_j succeeds X_{ik}^{j} (fig.6.7b). M_s will therefore be represented as a Backward edge w.r.t. RL_{ik} in the corresponding event graph. Once again this leads to a contradiction since the algorithm does not allow Backward edges to intersect recovery lines.

(iv) Violation of Property d: Let X_{ik}^{j} and X_{ik}^{g} be two checkpoints in RL_{ik}. If possible let $BDS(X_{ik}^{j},X_{ik}^{g})$ be non null. This implies that there is at least a message M_s sent before X_{ik}^{g} which is received after X_{ik}^{j} and has not been saved in RL_{ik} (fig.6.7c). From the algorithm it can be observed that since M_s was sent before checkpoint X_{ik}^{g}, the entry corresponding to P_i in the checkpoint vector of M_s will contain a value y which is $< k$. This message M_s is received by P_j after X_{ik}^{j} has been established. Therefore M_s will be associated with recovery lines RL_{iu} where $y < u \leq k$, assuming without loss of generality that X_{ik}^{j} is currently the latest response checkpoint in P_j. i.e. M_s will be saved in RL_{ik} and this leads to a contradiction.

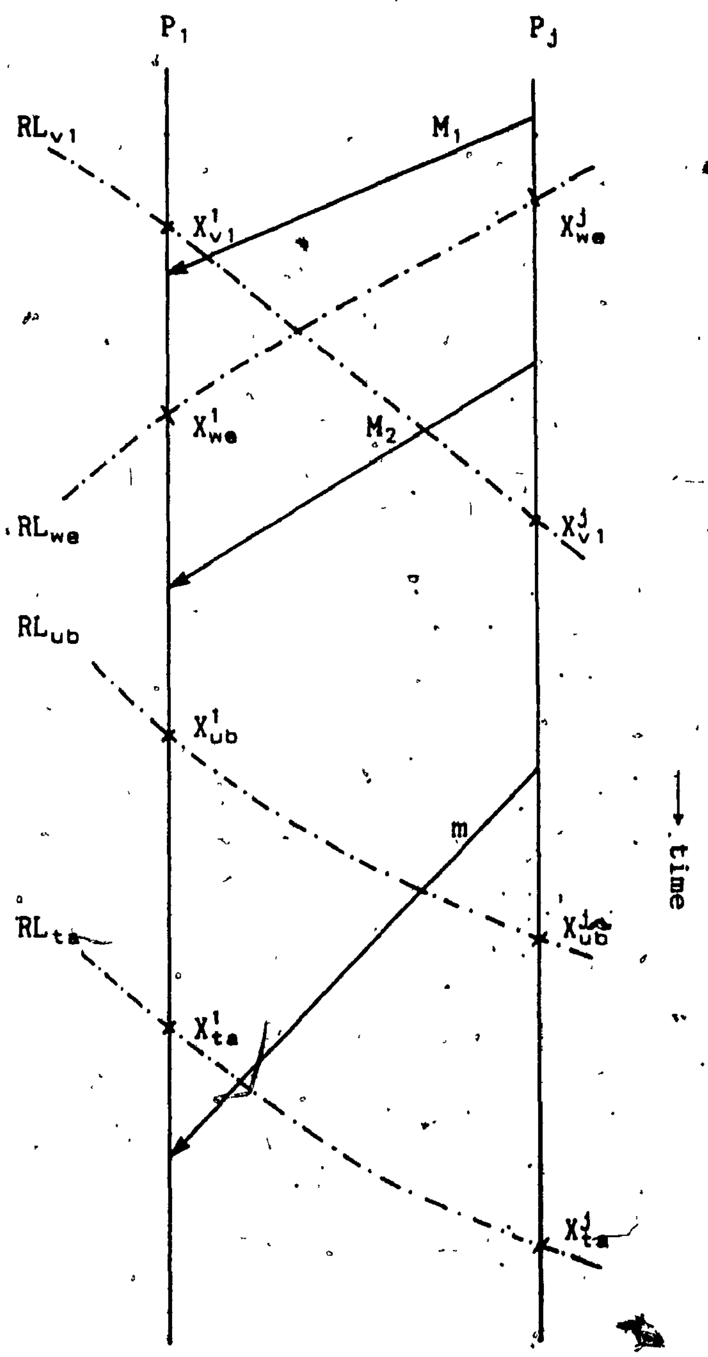
Part 2: Any fault (F) between two self induced checkpoints with ordinal numbers k and (k+1) in a process P_i, will be overcome by the algorithm by rolling back P_i to the kth checkpoint and the other processes to the corresponding response checkpoints if any. Avalanche of rollbacks (domino effect) can occur while overcoming the effect of the fault F if one of the processes say P_j will have to be rolled back to a checkpoint X^j_{ry} which precedes X^j_{ik}, as this in turn could lead to rollbacks of some other processes, eventually leading to an avalanche of rollbacks. This would only happen if on rolling back P_j to X^j_{ik} it was found that either an effect of fault F was not overcome or BDS or RDS was not null. This has shown to be impossible (Part 1). Thus the rollback is domino free. Incidentally, it is observed that the "unnecessary rollback", which is the portion of the computation in each process between the first effect of the fault in the process and the checkpoint to which the rollback occurred, in the worst case, is the computation in the region bounded on one side by the recovery line $\langle i,k \rangle$ and on the other by $\langle i,k+1 \rangle$ as shown in fig.6.7d.

Q.E.D.

6.3.3 Multiple Simultaneous rollback

Avizienis et.al [3] observe that, in large distributed systems, the probability of multiple errors occurring simultaneously in different processes is significant. Then it is quite possible that recovery and rollback can be initiated by a process while some other processes may have also initiated rollback.

Consider the example shown in fig.6.8. Let P_1 detect an error at time t_1 and decide to rollback to RL_{v1}. Before this recovery action has been completed in the system, some other process P_r decides to rollback



X^1_{v1} and X^j_{we} are the earliest checkpoints of P_1 and P_j corresponding to the set S.

Figure 6.8: Processes subjected to Multiple simultaneous recovery

to RL_{we}. Examining fig.6.8, we can state that processes P_1 and P_j should be effectively rolled back to X^1_{v1} and X^j_{we} respectively as the correct recovery. Only message M_1 should be inserted to the head of $channel_{j1}$ as message M_2 will be regenerated by P_j during re-execution from X^j_{we}. This informal discussion can be generalized to account for multiple simultaneous rollback in which the single recovery action will be a special case.

Let a system P of processes $(P_1,P_2,...P_n)$ be currently under rollback and recovery induced by a subset of these processes which have chosen a set $S:(RL_{va},RL_{wb},...)$ of recovery lines. We define a hypothetical Effective Recovery Line:

Definition: Effective Recovery Line (ERL):

This is defined as the maximal common prefix of the set $Z: \{p_{va},p_{wb},...p_{zm}\}$ of prefixes representing the system S of recovery lines $\{RL_{va},RL_{wb},...RL_{zm}\}$ to which concurrent rollback is in progress. That is,

$$p_{ERL} \preceq_{\pi} p_{va}; (p_{ERL} \preceq_{\pi} p_{wb},... \; p_{ERL} \preceq_{\pi} p_{zm});$$

and $\nexists p'$ s.t. $|p'| > |p_{ERL}|$ and $(p' \preceq_{\pi} p_{va}, p' \preceq_{\pi} p_{wb},...,p' \preceq_{\pi} p_{zm})$.

Alternatively, the ERL for the set S, is specified by a pair $\langle EX^S,EM^S\rangle$ where:

(a) EX^S is the set of checkpoints $(EX^S_1,EX^S_2,...EX^S_n)$ such that EX^S_j is the earliest checkpoint of process P_j in the set of checkpoints $(X^j_{va},X^j_{wb},...)$ of P_j corresponding to the chosen set of recovery lines S.

(b) EM^S is that subset of messages of $(\bigcup_S M^i)$ which intersect the Effective Recovery Line of S i.e. a message sent from P_q to P_r will be in EM^S if it was sent by P_q before EX_q^S and received by P_r after EX_r^S. An example of Multiple Simultaneous Rollback is given in Appendix B.

With respect to ERL the following assertion can be made.

Lemma 6.1: The ERL as defined above is a Valid recovery line.

Proof: Let ERL of the system S of chosen recovery lines $\{RL_{va}, RL_{wb}, \ldots RL_{zm}\}$ be represented by the prefix p_{ERL}. The proof is based on contradiction. If possible let p_{ERL} not be a Consistent prefix. This implies that there is a event $e_j \in p_{ERL}$ one of whose predecessors events $e_i \notin p_{ERL}$. Since $e_j \in p_{ERL}$, it must be common to the prefixes $\{p_{va}, p_{wb}, \ldots p_{zm}\}$ corresponding to the recovery lines of S, while e_i must not be present in at least one of the prefixes, say $p_{rk} \in Z$. As $e_j \in p_{rk}$ and $e_i \notin p_{rk}$, this implies that p_{rk} itself is not a Consistent prefix. Thus giving raise to a contradiction.

Q.E.D.

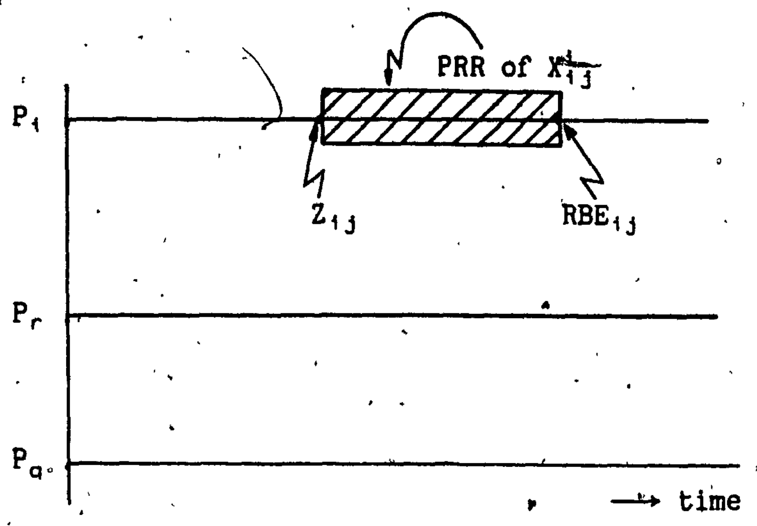
6.4 Discussion on the Rollback algorithm

The Rollback algorithm has been presented in fig.6.6. In the following discussions we establish the correctness of the algorithm. This algorithm does not require freezing the execution of any process, and multiple processes may simultaneously initiate rollback recovery.

Every correct rollback scheme must effectively replace the rolled back segments of the computation in all affected processes by Nop's i.e. no effect of these computation segments must persist after the rollback. Distributed systems consists of two parts namely processes and channels. Therefore for correct rollback, the events in the rolled back segment must neither affect the state of the processes nor affect the channels. In order to precisely state the behavior of the system during rollback the following notations are introduced.

Definition: Primary Rollback Region (PRR)

If the rollback event rolls back the process to a self induced checkpoint (either because of directives from the error diagnosis module, or because of receipt of a rollback message specifying this checkpoint) then the computation segment between the chosen self induced checkpoint and the rollback event is known as the Primary Rollback Region. PRR_{ij} denotes the Primary Rollback Region of Process P_i associated with its j^{th} self induced checkpoint (fig.6.9a). Boundaries of this region are dynamically defined, as the end of the region is defined by the Rollback event, and the beginning of the region is defined by the corresponding Checkpoint event.



Legend: RBE_{ij} – Rollback Event which rolls back process to RL_{ij}

Figure 6.9a: Primary Rollback Region

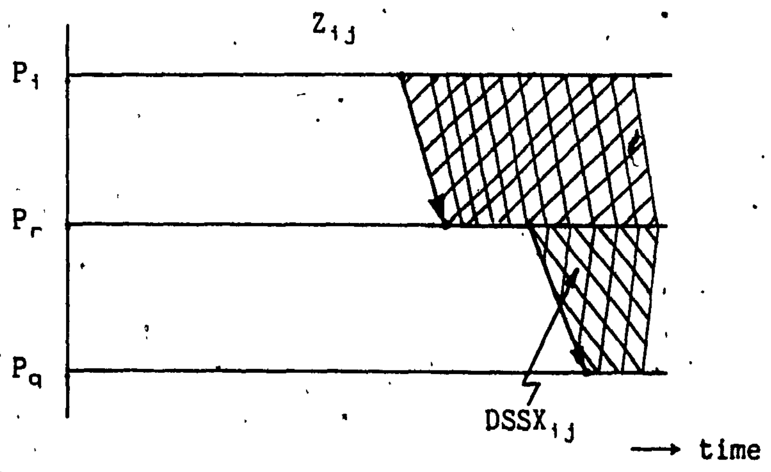


Figure 6.9b: Descendant Set

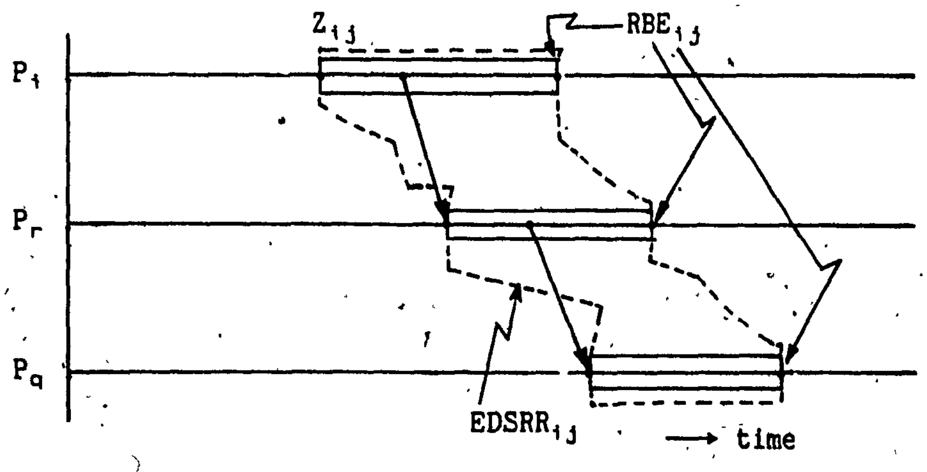


Figure 6.9c: Exclusive Descendant Set

Definition: Descendent Set of a Self induced Checkpoint ($DSSX_{ij}$)

This is the set of all events in the system which are either direct or indirect descendents (i.e. successors) of the checkpoint recording event X^{i}_{ij}. Equivalently, there exists paths from X^{i}_{ij} to each of the candidate events in the graphical representation of the computation (fig.6.9b).

Definition: Exclusive Descendent Set of PRR_{ij} ($EDSRR_{ij}$)

This is the set of those events in $DSSX_{ij}$ which are not events executed during the re-execution commencing from X^{k}_{ij}, k=1...n or any descendent of these events (fig.6.9c).

Definition: Rollback Region (RBR)

The segments of the computation in each process between the checkpoints of a rollback line and the corresponding rollback event in the process, jointly define a Rollback Region for that Recovery line. It should be noted that all events in the Rollback Region of a Recovery line RL_{ij} belong to $EDSRR_{ij}$, and is denoted as RBR_{ij}. Also the projection of RBR_{ij} on process P_i will be PRR_{ij}. The projection of RBR_{ij} on a process P_r will be denoted as RBR^{r}_{ij}.

Definition: Faulty message

Every message generated inside a rollback region (RBR) is defined to be a faulty message, and the others are defined to be nonfaulty.

Definition: Effectively Forwarded message

A message M is Effectively Forwarded to a process P_j in the system of processes P provided M does not get revoked in the rollbacks induced by S. Such messages are also called Valid messages.

Definition: Prerollback Messages

Prerollback messages are those messages which are generated by a process before its first rollback triggered by S occurs.

Definition: Pending Messages

Pending messages are those Prerollback messages which have not been accepted before the receiver commences its rollback due to S.

Definition: Newly Generated Messages

Newly generated messages are those messages which are generated during re-execution following the first rollback, of the sending process, due to S.

6.4.1 Correctness of the Rollback algorithm

In a system subjected to concurrent rollback to the set S:$\{RL_{va}, RL_{wb}, \ldots RL_{zm}\}$ of recovery lines, a functionally correct Rollback Recovery algorithm should undo all of the effects of every event in the corresponding rollback regions, on the system. Undoing an intraprocess event requires annulling the event's effect on the state of the process, so that the process assumes the state it had before the event was executed.

Undoing a Send event requires revocation of the message it had sent, and also annulling the consequent effect on the receiver process. The sender process must assume the state it had before the Send event was executed. Undoing a Receive event implies that the message received by this event is returned to the channel from which it was received, and the process assumes the state it had before the Receive event was executed. In this context a correctly functioning Rollback Recovery algorithm must perform the following tasks:

(a) Undo all local effects on the process state produced by events belonging to any RBR of S, that were executed by this process.

and

(b) Undo all effects on the Channel states produced by events $\in$ any RBR of S by:

(1) Loading messages whose Receive events $\in$ RBR of S but whose corresponding Send events $\notin$ any RBR of S to the head of the respective channels.

(2) Discarding all those messages whose Send events $\in$ RBR of S from the set of Pending and Newly Generated messages.

(3) Preserving the sequence of the messages in the channels so that during re-execution the messages will be received in the order in which they were generated.

The correctness proof of the proposed Rollback recovery algorithm is presented as a series of Lemmas and Corollaries based on the following additional assumptions and observations.

Additional Assumption:

(BA1): Individual events are atomic. The receipt of a rollback message and subsequent execution of the induced rollback event (if at all) is performed atomically.

Observations:

(Obs1): RBR and EDSRR of a self induced checkpoint are defined only on execution of the corresponding Rollback event. A process executes the Rollback event and rolls back to a recovery line either on receipt of a directive from the Error diagnosis module, or on receipt of the first rollback message specifying this recovery line.

(Obs2): During rollback of a system to a recovery line RL_{kx}:

- Exactly one rollback message $R\langle k,x\rangle$ traverses each channel of the system.
- A process P_j sends this rollback message $R\langle k,x\rangle$ on all its output channels, when it initiates rollback to X^j_{kx}.
- Eventually all processes will receive this rollback message $R\langle k,x\rangle$ on all of their input channels.

(Obs3): Process P_i on receiving the rollback message $R\langle k,x\rangle$ for the first time will perform the following:

P_i rolls back itself to X^i_{kx} and introduces messages associated with X^i_{kx} to the head of its respective input channels provided X^i_{kx} exists. Puts all channels except the one on which $R\langle k,x\rangle$ was received, to the Cautious state and enters $R\langle k,x\rangle$ into their ARM lists. Sends rollback message $R\langle k,x\rangle$ on all its output channels. It then resumes execution of the application. A channel in the Cautious state will discard all those messages which depend on any of the recovery lines entered in the ARM list of that channel. A channel will delete an entry from its ARM list on receiving the corresponding rollback message. The channel will be switched back to the normal state when its ARM list becomes empty.

Lemma 6.2: All local effects on the process state produced by events $\in$ any RBR of S, that were executed by this process are undone.

Proof: Based on assumption BA1, it can be inferred that the rollback regions in each process due to concurrent rollback are temporally mutually exclusive. Therefore an event (say E_1) can belong to only one rollback region at any time.

Consider one such event $E_1 \in RBR_{ix}$. Let E_1 not be undone properly during rollback, i.e. its effect still persists. But this is impossible since the retrieved system state is the state that was saved at the beginning of this RBR on stable storage and hence will not reflect the influence of E_1.

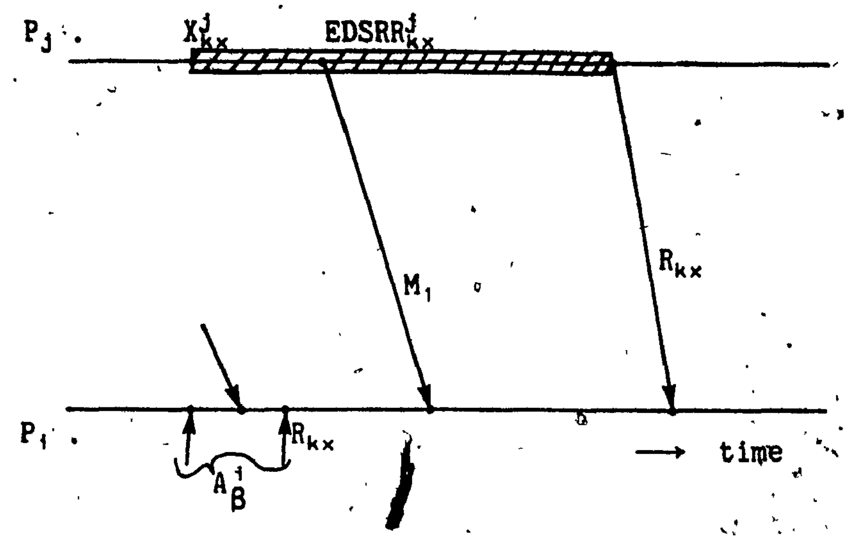
From the above it can be inferred that the effect of every event in a recovery region will be undone. Additionally from observations Obs1 and Obs2 it is known that every process will eventually receive all rollback messages corresponding to S and will undo all events in the corresponding rollback regions if it exists. Thus proving the assertion.

Q.E.D.

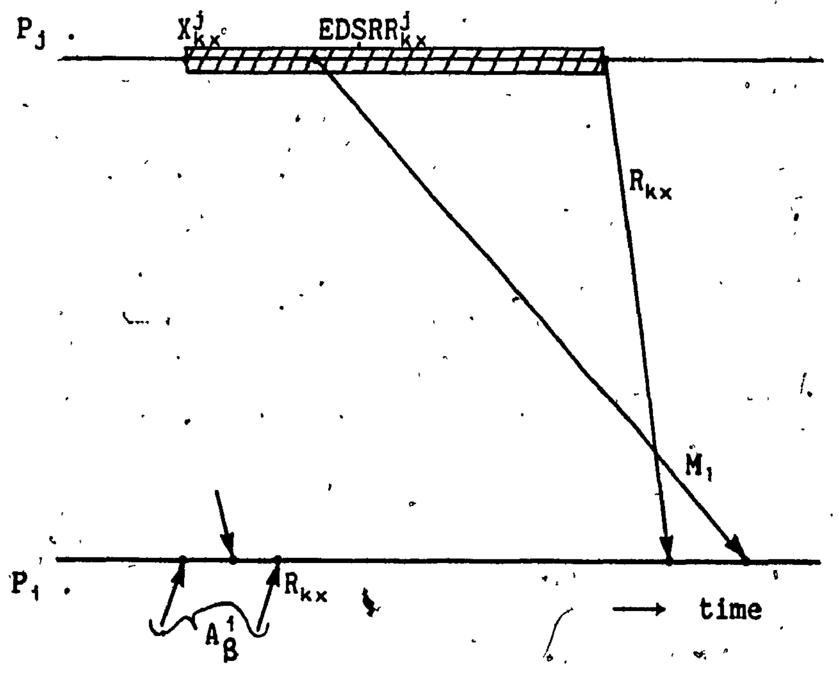
Lemma 6.3: No message generated in a rollback region will be Effectively Forwarded.

Proof: Assume the system is being subjected to concurrent rollback to the set S: $\{RL_{va}, RL_{wb}, \ldots RL_{zm}\}$ of recovery lines. Let $|S|$ be α. Consider a message M_1 generated in a rollback region RBR_{kx}. It could either have been generated in PRR_{kx} or in some RBR^{j}_{kx}, where $j \neq k$. Assume that M_1 was generated by P_j in RBR^{j}_{kx}, and let it be effectively forwarded to process P_i, where $RL_{kx} \in S$. Before M_1 is received, let P_i have received β rollback messages belonging to S, and let it receive the remaining $(\alpha\text{-}\beta)$ rollback messages subsequently. Let A^{i}_{β} and $A^{i}_{\alpha\text{-}\beta}$ represent these two sets of rollback messages. Recall that $R\langle k,x \rangle$ represents the Rollback message corresponding to RL_{kx}. As $R\langle k,x \rangle$ could belong either to A^{i}_{β} or $A^{i}_{\alpha\text{-}\beta}$ (fig.6.10) the following two cases have to be considered.

<u>Case 1</u> $R\langle k,x \rangle \in A^{i}_{\beta}$: This implies that $R\langle k,x \rangle$ is received before M_1 is received. Based on observation Obs3, if M_1 is effectively forwarded then

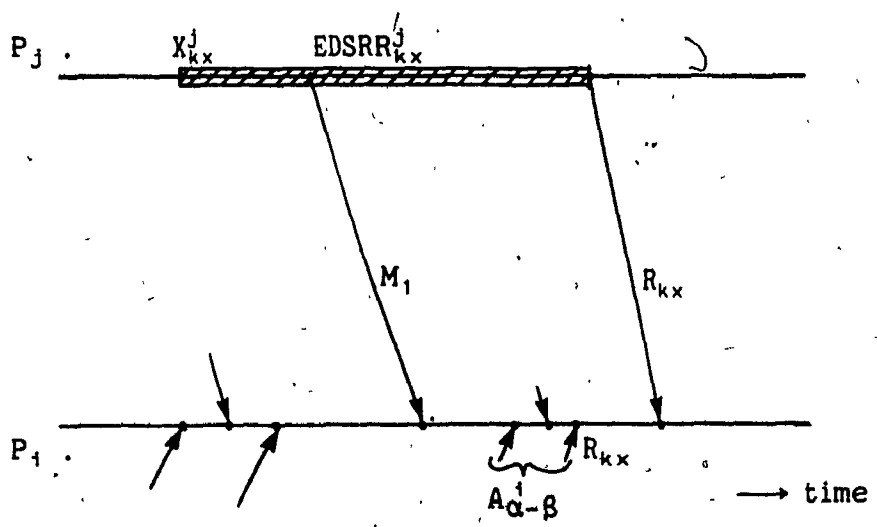


Case 1(i): $R\langle k,x\rangle \in ARM_{ij}$ when M_1 is received.



Case 1(ii): $R\langle k,x\rangle \notin ARM_{ij}$ when M_1 is received

Figure 6.10: Lemma 6.3-Messages generated in EDSRR (contd.)



Case 2: $R_{kx} \in A^i_{\alpha-\beta}$

Figure 6.10: Lemma 6.3-Messages generated in EDSRR

Send(M_1) in $P_j \prec$ Every Checkpoint in P_j corresponding to the rollback messages in ARM_{ji}. Then either

(i) $R\langle k,x\rangle$ is in ARM_{ji}, in which case the above condition requires Send(M_1) $\prec X^j_{kx}$, i.e. M_1 could not have been generated in $EDSRR^j_{kx}$. This leads to a contradiction.

(ii) $R\langle k,x\rangle$ is not in ARM_{ji} when M_1 is received. This implies that $R\langle k,x\rangle$ has been received on Channel$_{ji}$ before M_1 is received. But this is impossible because the channel is FIFO and M_1 is sent before $R\langle k,x\rangle$ on the channel. Thus leading to a contradiction.

<u>Case 2</u> $R\langle k,x\rangle \in A^i_{\alpha\text{-}\beta}$: This implies that $R\langle k,x\rangle$ is received by P_i after it receives M_1. If M_1 should be effectively forwarded, then the corresponding Send event should never be undone by P_j. Clearly then Send(M_1) $\prec X^j_{kx}$, this results in a contradiction as M_1 has been assumed to be generated in $EDSRR^j_{kx}$.

Similar arguments hold good even when Send(M_1) $\in$ PRR_k[illegible]

Q.E.D.

Lemma 6.4: No nonfaulty message gets discarded by the rollback algorithm.

Proof: If possible let M_1, a nonfaulty message sent by P_i to P_j, be discarded by the rollback algorithm. Since M_1 is discarded either

<u>Case 1</u>: It was received on Channel$_{ji}$ when ARM_{ji} was expecting at least one rollback message $R\langle k,x\rangle$ such that $X^j_{kx} \prec$ Send(M_1). This implies

that M_1 was generated in a rollback region possibly in RBR^j_{kx} and hence is faulty. Thus resulting in a contradiction.

Case 2: The Receive event which accepted M_1 was undone due to receipt of a rollback message say $R\langle k,x\rangle$ and $X^j_{kx} \prec$ Send(M_1). (Note if Send(M_1) $\prec X^j_{kx}$, the message M_1 is reinserted on the channel and hence is not discarded). This once again requires M_1 to have been generated in RBR^j_{kx} thus giving raise to a contradiction.

Similar arguments hold good when M_1 is generated by PRR_{ry}.

Q.E.D.

Lemma 6.5: Every Prerollback message whose Receive event $\in$ RBR but whose Send events $\notin$ any RBR of S, will be Effectively Forwarded during re-execution following rollback.

Proof: Consider a message M_1 whose Receive event in $P_i \in RBR_{ky}$ of S, and whose Send event in $P_j \notin$ any RBR of S. This implies that the Receive event $\notin$ prefix B_{ky} of the computation, representing the Recovery line RL_{ky}. Since M_1 is a Prerollback message and Send $\notin$ any RBR of S, the Send event $\prec$ Every checkpoint in P_j corresponding to S. Therefore Send(M_1) $\prec X^j_{ky}$ which implies that Send(M_1) $\in B_{ky}$. As Send $\in B_{ky}$ but Receive $\notin B_{ky}$, the Message M_1 will intersect RL_{ky}, and from the algorithm it is seen that M_1 will be reinserted to the head of the channel when the receiver rolls back to RL_{ky}.

Assume that the reinserted Prerollback message M_1 is not Effectively Forwarded during re-execution. This implies that the message was revoked by its sender P_j. A Sender process revokes a message only if the Send event $\in$ a RBR of S. Thus giving raise to a contradiction.

Q.E.D.

Lemma 6.6: No Prerollback message which has already been received and whose Receive event $\notin$ any RBR of S, will be reaccepted during re-execution following rollback.

Proof: If possible let a Prerollback message M_1 be Effectively Forwarded during re-execution following rollback to a recovery line $RL_{ky} \in S$. If M_1 should be Effectively Forwarded during re-execution then its Send event $\in$ Computation prefix B_{ky} : representing this recovery line, and its Receive event $\notin B_{ky}$. This is because by Lemma 6.3, M_1 cannot have been generated in any RBR. As M_1 has already been received during the original computation, the above requirement on the Send and Receive events implies that the Receive event $\in$ RBR. Thus raising a contradiction.

Q.E.D.

Corollary 6.1: Only those Pending & Newly generated messages whose Send events $\in$ any RBR of S, will be discarded.

Proof: Two possibilities exist and have to be considered:

Case 1: If possible let a Pending or Newly generated message M_1 whose Send event $\notin$ any RBR of S, be discarded. By definition a message whose Send event $\notin$ any RBR, is a nonfaulty message. By lemma 6.4 such a message cannot be discarded and thus leads to a contradiction.

Case 2: If possible let the Pending or Newly generated message M_1 whose Send event $\in$ a RBR of S, not be discarded. By definition since the Send event $\in$ a RBR the message must be a faulty message and by lemma 6.3 a contradiction results.

Q.E.D.

Lemma 6.7: The messages that are effectively forwarded on each channel will be in the sequence in which they were generated by the Sender.

Proof: As far as the pending and the new messages are concerned it is easily seen that the sequence is not affected. This is because the channel is FIFO and if a pending message in the channel is discarded then all its successors will also be discarded. The sequence of undiscarded messages is in no way altered. Additionally, as new messages are added only to the tail of the channel they will not affect the sequence.

Now consider the case in which messages are inserted to the head of the channel due to rollback of the receiver. This is the only possible action which can disrupt the sequence. Consider two messages m_a and m_b which are reinserted by process P_i on its input channel from process P_j. Let m_b be inserted ahead of m_a, whereas m_a was generated ahead of m_b at the sender. During rollback, messages are reinserted only if they intersect the corresponding recovery lines. The following two cases must be considered.

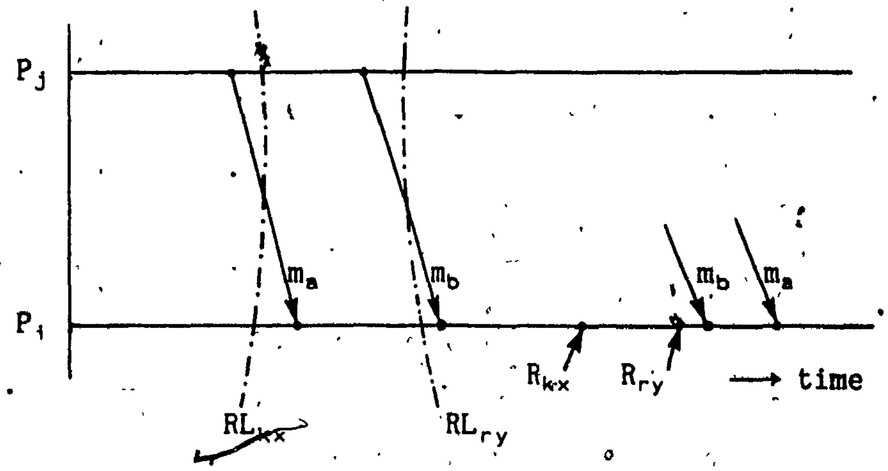
Case (a): m_a and m_b are reinserted due to rollback induced by a rollback message $R\langle k,x\rangle$.

From the rollback algorithm it is clear that if m_b was inserted in front of m_a in the channel, then the Receive(m_b) $\prec$ Receive(m_a). Since the channel is FIFO this implies Send(m_b) $\prec$ Send(m_a), thus giving raise to a contradiction.

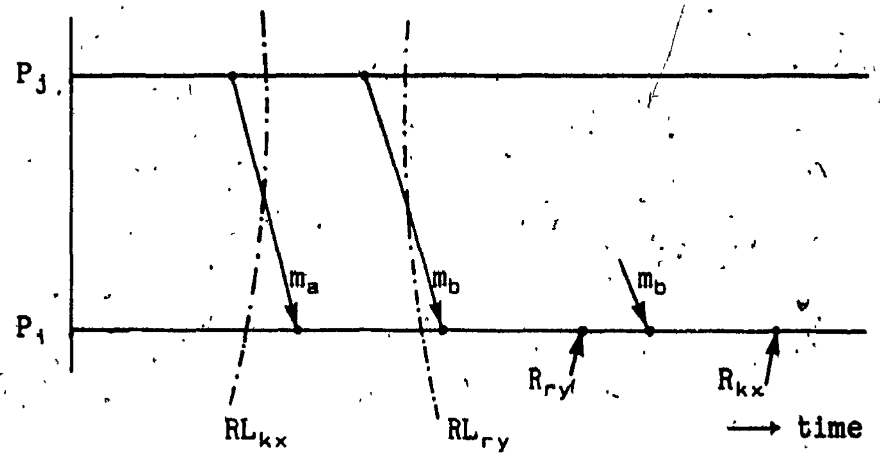
Case (b): m_a and m_b are inserted during rollback induced by two different rollback messages $R\langle k,x\rangle$ and $R\langle r,y\rangle$ respectively (fig.6.11a,b).

Consider the following temporal orders in which $R\langle k,x\rangle$ and $R\langle r,y\rangle$ are received.

(Order 1): $R\langle k,x\rangle \prec R\langle r,y\rangle$: Due to the receipt of $R\langle k,x\rangle$, message m_a is inserted to the head of the channel. Now if m_b should be inserted in front of m_a, the rollback message $R\langle r,y\rangle$ must be received before m_a is consumed, otherwise m_b and m_a will once again intersect the same



Case b(i): $R_{kx} \prec R_{ry}$



Case b(ii): $R_{ry} \prec R_{kx}$

Figure 6.11: Lemma 6.7: Message Sequence preservation

recovery line leading to case (a). Since in rolling back to $R\langle r,y\rangle$ message m_b must be reintroduced, this requires $X^i_{ry} \prec X^i_{kx}$, otherwise m_b cannot be introduced in front of m_a). Since m_a and m_b do not intersect a common recovery line, $X^i_{ry} \prec X^i_{kx}$ implies Receive(m_b) $\prec$ Receive(m_a). This requires Send(m_b) $\prec$ Send(m_a). Thus giving raise to a contradiction.

(Order 2): $R\langle r,y\rangle \prec R\langle k,x\rangle$: When the process rolls back to X^i_{ry} message m_b is inserted into the channel. If m_b should be ahead of m_a, this message m_b must be received before $R\langle k,x\rangle$ is received and additionally $X^i_{ry} \prec X^i_{kx}$, otherwise Receive(m_b) would also be undone. $X^i_{kx} \prec$ Receive(m_a) since m_a should intersect RL_{kx}. This as above leads to the requirement that Send(m_b) $\prec$ Send(m_a). Thus giving raise to a contradiction.

Q.E.D.

Corollary 6.2: The proposed rollback recovery algorithm is functionally correct.

Proof: From Lemmas 6.2 to 6.7 and Corollary 6.1, it can be readily seen that the algorithm meets the correctness criteria listed at the beginning of this section.

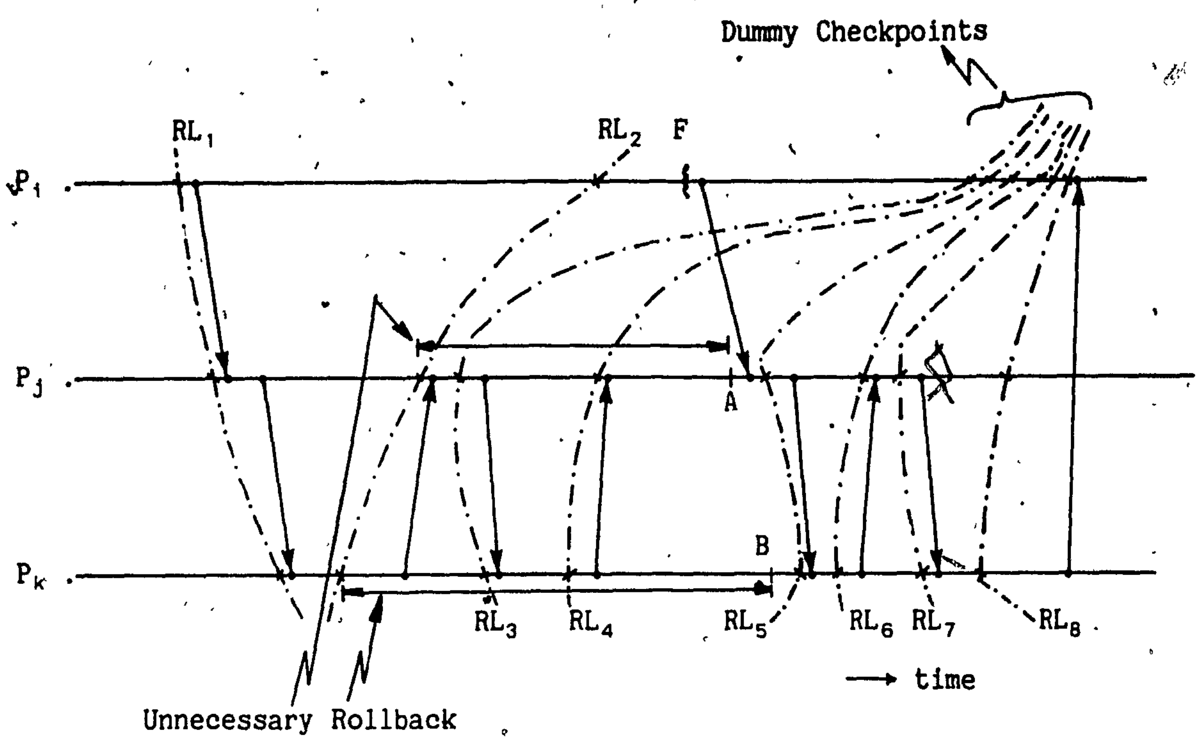
Q.E.D.

6.5 Related work and Comparison

In [54], Wood reports an unplanned checkpointing and recovery control protocol intended for a system of intercommunicating distributed processes. The two criteria addressed by his protocol are: reverting the system to a consistent state and supporting recovery point safety. He does not address the domino effects. In his case the processes are allowed to establish checkpoints asynchronously and no coordination is enforced between their checkpointing operations. At the time of rollback, the system searches for an appropriate recovery line and rolls back all affected processes to this recovery line. Due to the uncoordinated checkpointing the domino rollback problem could still exist in his case. However, there is little overhead incurred during the normal execution of the processes, as opposed to recovery time, which makes his method suitable for certain realtime applications.

As our algorithm is based on a planned and asynchronous strategy only such algorithms will be considered. Consequently, although Randell [38] has proposed a planned checkpointing strategy for domino-free rollback, as the algorithm is synchronous in nature it is not compared here.

Briatico et.al. [7] have proposed a distributed domino-free rollback protocol. In their scheme the processes are allowed to establish checkpoints at their own pace and are assigned ordinal numbers from a monotonically increasing counter which is incremented every time a checkpoint is created. Every process appends its latest checkpoint number with every application message it sends out. Upon receiving a message (M) a process P_j will check the received checkpoint number (s) with its own (r) (fig.6.12). If $s > r$ then (s-r) additional (dummy) checkpoints are forced on P_j. If on the other hand $s < r$ then the message (M) is associated with all checkpoints whose ordinal number (y) satisfies the relation $s < y \leq r$. The receiver appropriately updates its ordinal number counters. All checkpoints of P_j with numbers $\leq s$ are joined to corresponding checkpoints of P_i the sender i.e. the recovery lines are constructed by joining checkpoints with the same ordinal numbers (Iso-checkpoint number). Consider the case when $r \gg s$ and P_i is later required to rollback to its s^{th} checkpoint. This will force P_j to be rolled back to its s^{th} checkpoint. However, it would have been enough for the receiver (P_j) to be rolled back to its r^{th} checkpoint in order to undo the effects of this possibly contaminated message. Thus this algorithm results in unnecessary rollbacks. It has another drawback in creating excessive dummy checkpoints if the sender's checkpoint number is greater than the receiver's checkpoint number. These drawbacks are possibly caused by their labelling scheme which does not consider the temporal relationship among the interprocess interactions, unlike ours. In contrast, to Briatico's our scheme guarantees Minimal distance rollback by utilizing the recovery points efficiently while setting up the recovery lines. Moreover, our scheme does not create unnecessary checkpoints. However, to ensure minimal distance rollback, which is not ensured by Briatico et.al., our



Remarks:

(1) P_j and P_k are unnecessarily rolled back to their Checkpoints corresponding to RL$_2$ in order to overcome fault F in P_i. It would have been sufficient to rollback P_j and P_k to states A and B respectively.

(2) Lot of dummy checkpoints will be created when a process (P_i) which infrequently establishes checkpoints, infrequently communicates with a group of frequently communicating and checkpointing processes (P_j and P_k).

Figure 6.12: Exceptional cases for Briatico's Checkpointing scheme

scheme may induce more checkpoints than their scheme. The exact increase will depend on the frequency of checkpointing within the processes and the dynamics of interprocess message communication.

Koo [25] has proposed a rollback recovery algorithm recently. In this algorithm processes establish local checkpoints based on their requirements. Once a process takes a checkpoint, then it requests the other processes also to checkpoint. A process receiving this request will establish a new checkpoint only if its previous checkpoint does not account for all the causes whose effects have been felt before this checkpoint of the originator. Thus a Consistent global state is captured. Like Briatico's algorithm this scheme also does not track dependencies and hence could lead to unnecessary rollback in a situation similar to the one depicted in fig.6.12.

Strom et.al. propose a scheme for domino-free rollback [46]. Their strategy involves a slow speed stable store. Their model of DCS is not the same as ours. However, it can be observed that every message crossing an arbitrary recovery line is saved in their scheme and they also assume that computations are exactly repeatable.

In their latest work Kim et.al. [24] report a technique wherein distributed recovery blocks are executed aynchronously. Their checkpoints are coordinated to ensure domino-free and minimal distance rollback. They do not address the issue of multiple and simultaneous rollback. Moreover, they store all interprocess messages unlike in our case. It is interesting to note that the two independently developed schemes (ours

[50,37] and Kim's) have a similar approach to the coordination of checkpoints.

6.6 Summary

We have presented a checkpointing scheme which coördinates the creation of checkpoints in the processes based on inter-process interactions. Within itself a process may initiate checkpointing based on its own local requirements and environment. This algorithm minimizes rollback distance, saves messages selectively, does not require special control messages during checkpointing and does not create dummy checkpoints. Additionally, this algorithm purges only the contaminated messages while rolling back to the chosen recovery line. For rollback purposes a process would normally choose a checkpoint owned by itself. This is done so as to rollback just the right amount for undoing the computation contaminated by the error.

In large distributed systems multiple errors could concurrently occur. Our algorithm has been designed to work efficiently even in those environments where multiple processes initiate recovery actions concurrently. Then processes cooperatively define an Effective recovery line to which they asynchronously rollback and restart their execution.

Chapter 7

Conclusions

System-wide information is necessary for efficiently solving a number of problems in distributed systems. The exact information to be gathered and the coordination required between the processes while gathering this information depends on the applications. One of the main contributions of this thesis has been the evolution of a Global state classification scheme. Chapter 3 was devoted to this study. A number of problems requiring global information were analyzed and the Global states were classified into four distinct categories namely: Statistical Global states, Stable Global states, Consistent Global state and Synchronized Global state. Interesting properties of these states have been derived after studying generic problems. Stable global states have been proved to be able to detect Stable properties and the Global Minimum property. Consistent Global states have been found to posses properties of Reachability, Recoverability and Global invariant preservability. The Global states themselves have been shown to be interrelated. A Synchronized Global state $\Rightarrow$ Consistent Global state $\Rightarrow$ Stable global state $\Rightarrow$ Statistical Global State.

The problem of detection of Global states has been addressed. Message efficient Global state detection algorithms for detecting Statistical, Stable and Consistent Global states have been proposed. In the model used, there is at least one process which initiates and one process which compiles the global state. These algorithms have a message complexity of O(n); where n is the Number of processes in the system, and the algorithms have negligible local recording time. The algorithms achieve this performance by distributing the task of accounting for the messages in transit, between the receiver and sender processes. The algorithms have marker flags piggybacked on application messages in order to support Non-FIFO communication subsystems like prioritized message delivery and out-of-band signalling.

Application of the Global Minimum detection property of Stable global states to Distributed Discrete Event simulation has been examined. A new scheme has been proposed for updating local simulation clocks in process graphs having strongly connected components. The problem has been methodically analyzed and three different Optimization problems have been formulated. The search for an optimal solution for minimizing the total number of messages needed in order to update the clock has been shown to be NP-complete. Logically Synchronous heuristc solutions have been proposed for this problem. The performance of this solution for strongly connected components of process graphs has been shown to be better than the existing solutions. Subsequently, a hybrid scheme which dynamically switches between a synchronous update scheme and a asynchronous update scheme depending on the estimated clock jump has been proposed. The performance of this algorithm can be tuned by properly selecting the threshold value for performing the switch.

As an application of Consistent global states the problem of Backward Error Recovery of distributed systems was considered. The reasons for Domino Rollback were carefully analyzed and a new algorithm for achieving Domino free rollback has been proposed. This problem is an application of the recoverability property of Consistent Global states. A coordinated dependency tracking scheme has been designed for checkpointing the distributed computation. This scheme selectively stores messages for playback during re-execution. The rollback scheme has been designed to support multiple concurrent rollbacks which is highly probable in large distributed systems. The recovery takes place without having to freeze the application. The algorithm neither imposes restriction on intercommunication patterns nor requires periodic synchronization. This scheme minimizes the rollback distance by rolling back only the affected processes and choosing checkpoints in each process which were created in response to the Self induced checkpoint in the faulty process which just precedes that fault.

In short, in this thesis a Classification scheme for Global state has been proposed. Additionally, methods for efficiently gathering and managing the required Global information specific to the needs of two different applications namely: Distributed Discrete Event Simulation and Backward Error Recovery have been developed.

7.1 Suggestions for future work

A broad spectrum has been covered in this thesis. Consequently a number of interesting venues for continuation of this work can be suggested.

In the work on Global state detection, the model used in this thesis assumes specific processes for initiating and compiling global states. There are other interesting models in which specific initiators may be absent. For instance the model suggested by Liskov [31] for Distributed garbage collection assumes that all processes would take their own state whenever they want, and send it to a central site. This central site compiles the local recordings meaningfully to evolve a Stable Global state. In this model no specific process initiates a system wide recording. It will be interesting to develop Consistent Global State detection algorithms for these models. Deterministic algorithms for Consistent global state detection will require coordination between the processes during state recording. However, probabilistic algorithms could be examined in this context.

Debugging in distributed systems require a support facility for recording Consistent Global states. Generally on detection of an error, in order to be able to diagnose the cause of the error the error will have to be recreated. In this case the system will have to be rolled back to a previously saved Consistent global state and re-execution will have to be commenced from that state. During this run more detailed information which could aid in diagnosing the error can be gathered. An important aid to debugging will be to reconstruct the sequence of events that occurred in each of the processes. In this problem Dependency tracking of messages could help in identifying these precedences between events. This should be combined with Global state detection techniques to evolve a practical algorithm for reconstructing the Pomsets.

The General Global state detection algorithm proposed in Chapter 4, records global states in which the number of messages in transit on the channels can be determined. However, if the exact messages in transit are necessary i.e. just the number of messages is not sufficient, then these algorithms will have to be appropriately extended. The senders can keep a log of all messages transmitted by them and these senders will have to be informed about the messages accepted by the receivers before the receivers' state recording. Efficient ways for disseminating this information to the senders will have to be designed and analyzed.

In the context of Discrete Event Simulation we have developed heuristic algorithms and have shown that the upperbound is significantly lower than comparable algorithms. The exact tradeoff however is dependent on parameters like process structures, patterns of interaction during simulation and the magnitude of clock changes. In the hybrid algorithm additionally the threshold parameter is crucial for tuning the algorithm. An experimental study of practical problems would clarify the exact trade-offs and aid in tuning the algorithms for maximum efficiency.

The problem of Backward Error Recovery has been examined and solutions have been proposed for some important issues here. This work can be extended in several different directions. Based on the knowledge gained in solving the above problem we are convinced that adopting a strategy which requires playing back of old messages during re-execution is not good for Real time systems. This is because by the time the rollback to a previous checkpoint is performed the world might have altered significantly, and playing back these old messages may not be meaningful at all. This however depends on the process dynamics. A scheme for fault tolerance in Real time systems must address a number of issues starting with a model for Real time systems, support of periodic and aperiodic processes, scheduling and load balancing algorithms for meeting deadlines, process replication strategies, determination of the number of replicas required for achieving a certain degree of fault tolerance and other related issues. In this thesis it has been assumed that checkpoints once created are not discarded. Aspects worth investigating are the criteria for discarding checkpoints, optimal checkpointing strategies for our model, handling of suspicious messages, simplifications in the context of pseudo synchronous algorithms, supporting low speed stable storage, and coordinating shadow processes if any.

In conclusion the tip of the global state iceberg seems to have been examined in this thesis. Lot of issues remain to be solved.

References

1. M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery and F.B. Schneider, "Distributed Systems - Methods and tools for specification - An advanced course", Lecture notes in Computer Sci. No. 190, Springer-Verlag 1985.

2. T. Anderson and P.A. Lee, "Fault tolerance- Principles and Practice", Prentice Hall International 1981.

3. A. Avizienis and D.E. Ball, "On the achievement of a highly dependable and fault tolerant Air traffic control system", IEEE Computer Feb.87, pp. 84-89.

4. M. Banatre and F. Ployette, "A note on the design of a stable storage facility for Gothic", University of New Castle upon Tyne: TR-203 1985 pp70-73.

5. E.J. Berglund and D.R. Cheriton, "Amaze: A Distributed multi-player game program using the distributed V kernel", Proc. of the fourth conference on Distributed Computing Systems 1984, pp. 248-255.

6. G. Bracha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection", Proc. of the 1984 Symposium on Principles of Distributed Computing, pp. 285-301.

7. D. Briatico, A. Ciuffoletti and L. Simoncini, "A distributed domino -effect free recovery algorithm", Proc. Symposium on Reliability in Distributed Software and Data Base systems, Oct. 1984, pp207-215.

8. R.E. Bryant, "Simulation on a Distributed System", Proc. of the First conference on Distributed Computing Systems, 1979 pp. 544-552.

9. M.J. Carey, M. Livny and H. Lu, "Dynamic Task Allocation in a Distributed Database System", Proc. of the Fifth Intl. Conference on Distributed Computing Systems 1985, pp. 282-291.

10. K.M. Chandy and J. Misra, "A Non-trivial Example of Concurrent Processing: Distributed simulation", COMPSAC 78, pp. 822-826.

11. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determination of Global states of distributed systems", ACM Trans. on Computer Systems, Vol. 3, No. 1, Feb. 1985, pp. 63-75.

12. Y.F. Chen, A. Prakash and C.V. Ramamoorthy, "The Network Event Manager", Symposium on Computer Networking 1986, pp. 169-177.

13. D.R. Cheriton and W. Zwaenepoel, "Distributed Process groups in the V kernel", ACM Trans. Computer Systems, Vol. 3, No. 2, May 1985, pp. 77-107.

14. C.Y. Chin and K. Hwang, "Packet switching networks for multiprocessors and data flow computers", IEEE Trans. on Computers Vol. C-33,No. 11, 1984, pp. 991-1003.

15. R. Dechter and L. Kleinrock, "Broadcast communications and distributed algorithms", IEEE Trans. on Computers, Vol. 35,No. 3, March 1986, pp. 210-219.

16. E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations", Information Processing Letters, June 1983, pp. 217-219:

17. A.K. Elmagarmid, "A survey of distributed deadlock detection algorithms", ACM SIGMOD Record Vol. 15, No. 3, Sept. 1986, pp. 37-45.

18. M.J. Fischer, N.D. Griffith and N.A. Lynch, "Global states of a distributed system", IEEE Trans. on Software Eng., May 1982, pp. 198-202.

19. A.J. Frank, L.D. Wittie and A.J. Bernstein, "Group communication on netcomputers", Proc. of the Fourth conference on Distributed Computing Systems 1984, pp. 326-335.

20. H. Garcia Molina, F. Germano and W.H. Kohler, "Debugging a Distributed Computing System", IEEE Trans. on Software Eng. Vol. SE-10, No. 2, March 1984, pp. 210-219.

21. M.R. Garey and D.S. Johnson, "Computers and Intractability: A guide to the theory of NP-Completeness", W.H. Freeman and Company, San Francisco, 1979.

22. P. Jalote and R.H. Campbell, "Atomic actions for fault tolerance using CSP.", IEEE Trans. on Software Eng. Vol. SE-12,No. 1, Jan. 1986, pp.59-68.

23. K.H. Kim, "Approaches to Mechanization of the conversation scheme based on Monitors", IEEE Trans. on Software Eng. Vol. SE-8,No. 3, May 1982, pp. 189-197.

24. K.H. Kim, J.H. You and A. Aboulenaga, "A scheme for coordinated execution of independently designed recoverable distributed processes", Digest FTCS-16, 1986, pp. 130-135.

25. R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", IEEE Trans. on Software Eng., Vol. SE-13, No. 1, Jan. 1987, pp. 23-31.

26. K.B. Kumar and P. Kermani, "Analysis of a resequencing problem in communication networks", RC9767 IBM T.J. Watson Research Center, Yorktown Heights, NY, Aug. 1982.

27. T.H. Lai and T.H. Yang, "On distributed snapshots", Information Processing letters, Vol. 25, No. 3, May 29,1987, pp. 153-158.

28. S. Levitan, "Algorithms for broadcast protocol multiprocessors", Proc. of the Third conference on Distributed Computing Systems 1982, pp. 666-671.

29. H.F. Li, T. Radhakrishnan and K. Venkatesh, "Global state detection in Non-FIFO networks", Proc. of the Seventh conference on Distributed Computing Systems 1987.

30. A. Linton and F. Panzieri, "A communication system supporting large datagrams on a local area network", Software practice and experience, Vol.16 (3), Mar. 1986, pp. 277-289.

31. B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault Tolerant Distributed Garbage Collection", Proc. of the Fifth Annual Symp. on Principles of Distributed Computing Aug. 1986., pp. 29-39.

32. J. Misra, "Detecting Termination of Distributed Computations Using Markers", Proc. of the 1983 Symposium on Principles of Distributed Computing, pp. 290-294.

33. B.J. Nelson, "Remote Procedure Call", CMU-CS-81-119, Dept. of Computer Science, CMU, Pittsburgh, PA, 1981.

34. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel, "A network transparent high reliability distributed system", Proc. of the Eight symposium on Principles of OS, Asylomar, CA, 1981, pp. 169-177.

35. J. Postel, "User datagram protocol", RFC-768, USC/ Information Sciences Institute, Sept. 1980.

36. V. Pratt, "Modelling Concurrency with Partial Orders", International Journal of Parallel Programming, Vol. 15, No. 1.

37. T. Radhakrishnan, H.F. Li and K. Venkatesh, "Concurrent and Domino Free rollback recovery in distributed systems", Proc. of the IFIP conference on Distributed Processing, Oct. 5, 1987, Amsterdam.

38. B. Randell, "System Structure for Software fault tolerance", IEEE Trans. on Software Eng. SE-1, No. 2, June 1975, pp. 220-232.

39. R.L.D. Rees, "Computer System Reliability", INFOTECH State of the art report,1977

40. S.K. Sarin and N.A. Lynch, "Discarding Obsolete Information in a Replicated Database System", IEEE Trans. on Software Eng., Vol. SE-13, No. 1, Jan. 1987, pp. 39-47

41. J.H. Slatzer, D.P. Reed and D.D. Clark, "End-to-end arguments in system design", Proc. of the Second conference on Distributed Computing Systems 1981, pp. 509-512.

42. S.H. Son and A.K. Agrawala, "A Non-intrusive checkpointing scheme in distributed database systems", Proc. of the Fifteenth Conference on Fault Tolerant Computer Systems 1985, pp. 99-104.

43. M. Spezialetti and P. Kearns, "Efficient distributed snapshots", Proc. of the Sixth conference on Distributed Computing Systems 1986, pp. 382-388.

44. W. Stallings, "Data and Computer Communications", MacMillan, New York 1985.

45. J.A. Stankovic, K. Ramamritham and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-time Systems", IEEE Trans. on Computers, Vol. C-39, No. 12, Dec. 1985, pp. 1130-1147.

46. R.E. Strom and S. Yemini, "Optimistic recovery in distributed systems", ACM TOCS Vol. 3,No. 3, Aug. 1985, pp. 204-226.

47. K. Venkatesh and T. Radhakrishnan, "On a programming environment for multi-microcomputer systems", Proc. of the IEEE Intl. conference on Computers, Systems and Signal Processing, Bangalore, India, Dec. 84, pp. 275-278.

48. K. Venkatesh, "Time Advancement in distributed discrete event simulation", Internal report, Dept. of Computer Science, Concordia University, Montreal, Canada, May 1986.

49. K. Venkatesh, T. Radhakrishnan and H.F. Li, "Discrete event simulation in a distributed system", COMPSAC 86, Chicago, Oct. 1986, pp. 123-127.

50. K. Venkatesh, T. Radhakrishnan and H.F. Li, "Optimal Checkpointing and Local recording for Domino free rollback recovery", Information Processing letters, July 10,87, pp. 295-303.

51. R. Vickers and T. Vilmansen, "The evolution of Telecommunication Technology", Proc. of the IEEE, Sept. 1986, pp. 1231-1245.

52. S. Vinter, K. Ramamritham, D. Stemple, "Recoverable Actions in Gutenberg", Proc. of the symposium on Distributed Computer Systems, May 1986, pp. 242-249.

53. Y.T. Wang and R.J.T. Morris, "Load Sharing in Distributed Systems", IEEE Trans. on Computers Vol. C-34, No. 3, Mar. 1985, pp. 204-217.

54. W.G. Wood, "Recovery control of communicating processes in a distributed system", Reliable Computer Systems Ed. S.K. Shrivastava, Springer- Verlag 1985, pp. 448-484.

Appendix A

CLA algorithm for global state detection

First, we briefly review the CLA algorithm due to Chandy and Lamport through the same example used in §4.2. Suppose once again station "A" wishes to know the distribution of wagons (global state) in the system. How should "A" proceed? For convenience, assume all wagons are black and apply the CLA to the distributed system. As before, the state of a track (channel) is defined as the number of black wagons in transit and the state of a station (process) is the number of black wagons in the stations. CLA also assumes the availability of special "Red" wagons (Marker). In addition CLA assumes that no wagon will overtake another on a track, and expects the system to be accident free.

Station "A" initiates the global state recording by counting the number of black wagons presently in "A" and sends a "Red" wagon (marker) on each of its output tracks, AB and AC. It also appoints "observers" for its input tracks BA and CA, in order to count the number of black wagons which arrive afterwards. The above set of operations namely counting the number of black wagons, sending the red wagons and the appointment of observers is considered to be indivisible or atomic. Upon receiving the first red wagon, station "B" (or "C") counts the number of black wagons currently in the station. Then it sends a "red" wagon on each of its output tracks and appoints "observers" on its input tracks to count the number of black wagons which arrive afterwards. Subsequently, when a station ("A" or "B" or "C") receives a red wagon on any of its input tracks the observer of that track freezes his count. A station stops its state recording activity after it has received a red wagon on each of its input tracks. This locally recorded information is returned to station "A". A global state is formed when station "A" has received the local state information from all stations. The procedure is non-intrusive i.e. black wagons are still traversing the tracks during state recording. But the collected information does not constitute an instantaneous snapshot. As we will prove next the CLA algorithm actually records a Consistent global state though they have proposed it in the context of stable property detection.

In summary, the CLA assumes derailments (accidents) and overtaking of wagons never occur. It should be noted that the "red" wagons are overheads and should be minimized.

Theorem A.1

The CLA algorithm records a Consistent global state.

Proof: Assume the state S^* recorded by CLA is an Inconsistent global state. This could arise only if (1) the state of some process P_i is not recorded in S^*, or (2) a message which is in transit is not recorded, or (3) a message sent by a process P_i after its recording, has been received by P_j before its recording.

Case (1) State of P_i is not included in S^*: An iteration in CLA does not terminate until a marker traverses each channel in the system and a process records its state as soon as it receives the first marker of the iteration. So if P_i is reachable from the initiator through some input channel to P_i, eventually the state of P_i must be recorded in S^*. The stated possibility cannot arise.

Case (2) A message M which was in transit on channel$_{ij}$ is not recorded by P_j: Therefore M was received after the recording of channel$_{ij}$ had terminated. This implies that M was received after the marker message on channel$_{ij}$ and therefore it must have been sent on the channel after P_i recorded its state, giving rise to a contradiction again.

Case (3) A message M sent by P_i after its recording, is received by P_j before its recording.: Message M must have been received before any marker message is received by P_j. But this is impossible as M must follow the marker sent by P_i, immediately after its state recording, on channel$_{ij}$ and the channel is FIFO in nature.

Q.E.D.

So the state recorded by CLA corresponds to a Consistent cut (which will not contain backward channel edges). Thus CLA is capable of solving Consistent global state problems rather than just Stable ones.

Appendix B

Illustration of Checkpointing and Rollback operations

Checkpointing Scheme

Consider a distributed system of three processes. Let the STM representation of an instance of a distributed computation be as shown in fig.B.1. All messages shown in the figure are Application messages, since no control messages are necessary during the checkpointing phase. Significant aspects of the algorithm have been presented in the example. The computation is split into six segments and their significance is given below:

<u>Segment 1</u>

Each process establishes its first Self induced checkpoint and due to message interaction Response checkpoints are setup in the other processes. Focus on process P_2.

- P_2 creates its Self induced checkpoint and its CCP_2 = (-,0,-).
- When P_2 receives message M_1 with RCP = (-,-,0)

  Since $CCP_2(2)$ = 0 & RCP(2) = -;

      Attach M_1 to Checkpoint (-,0,-);

  Since $CCP_2(3)$ = - & RCP(3) = 0;

      Create Response Checkpoint (-,0,0);

  New CCP_2 = (-,0,0)

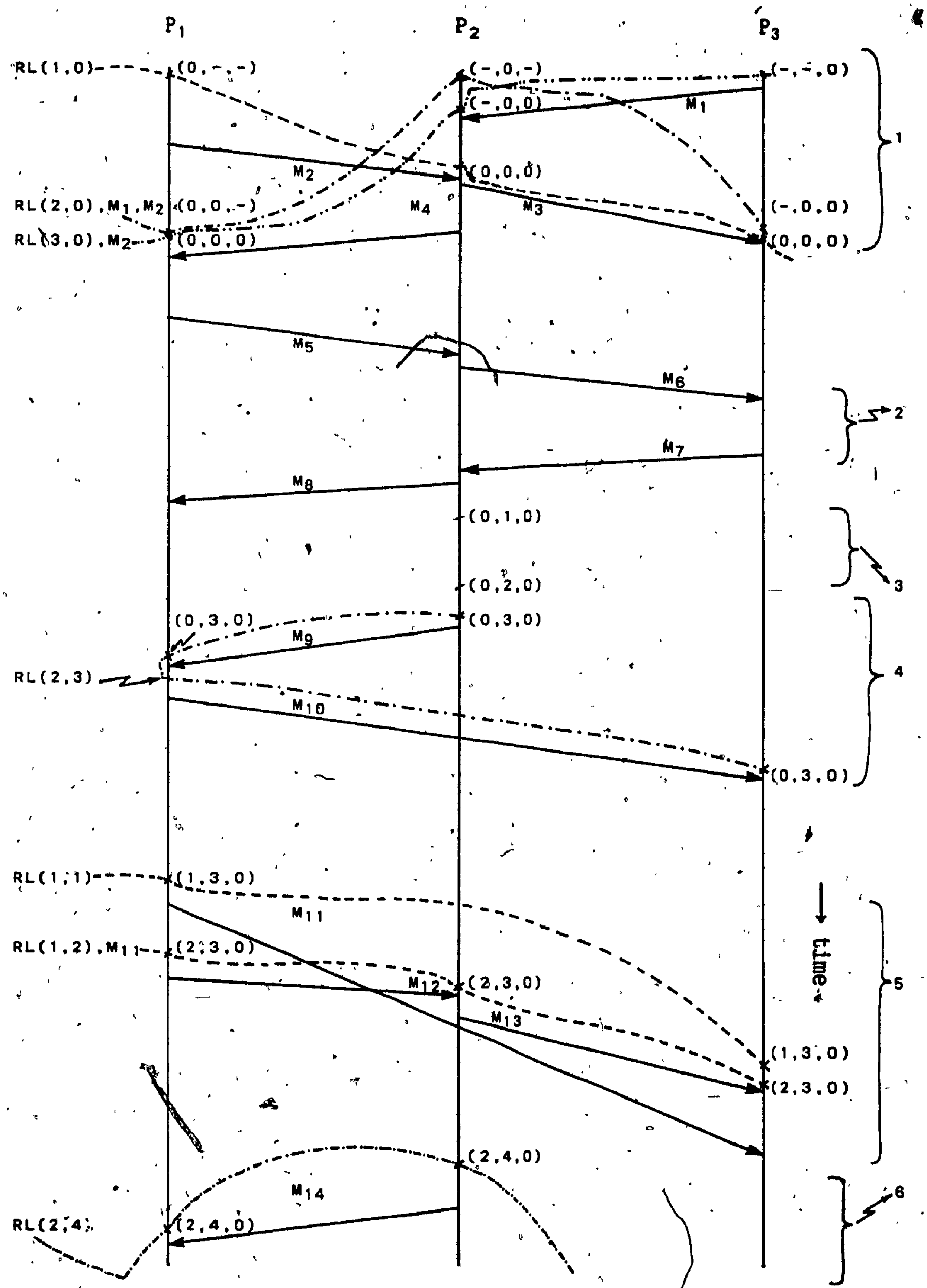


Figure B.1: Illustration of the Checkpointing algorithm

- When P_2 receives message M_2 with RCP = (0,-,-)

  Since $CCP_2(1)$ = - & RCP(1) = 0;

    Create Response Checkpoint (0,0,0)

  Since $CCP_2(2)$ = 0 & RCP(2) = -;

    Attach M_2 to Checkpoints (-,0,0) and (-,0,-).

  New CCP_2 = (0,0,0)

Similar functions occur in P_1 and P_3 also. Notice that when M_3 is received by P_3, it creates two response checkpoints one belonging to RL_{20} and the other to RL_{10}.

Segment 2

Before this segment all processes have CCP value of (0,0,0). In this segment no new Self induced checkpoints are created. Hence no new response checkpoints will get created. This is how the algorithm limits the number of checkpoints.

Segment 3

Self induced checkpoints (0,1,0), (0,2,0) are created by P_2. Since no interaction with the other processes occurs, no corresponding response checkpoints are created.

Segment 4

Shows how recovery lines get created because of indirect dependencies.

Segment 5

Illustrates the Copy Checkpoint phenomenon. Focus on process P_3. At the beginning of the segment $CCP_3 = (0,3,0)$

- P_3 receives message M_{13} with $RCP = (2,3,0)$

  Since $CCP_3(1) = 0$ & $RCP(1) = 2$;

  Create Response Checkpoint (2,3,0)

  New $CCP_3 = (2,3,0)$

- P_3 receives message M_{11} with $RCP = (1,3,0)$

  Since $CCP_3(1) = 2$ & $RCP(1) = 1$;

  Attach M_{11} to Checkpoint (2,3,0);

  Since Checkpoint X_{11}^{3} does not exist create response checkpoint (1,3,0) by copying Checkpoint (2,3,0) but deleting M_{11} from its message list. Note two Recovery lines owned by the same process do not intersect each other.

Segment 6

P_2 creates a Self induced checkpoint (2,4,0). When P_1 receives message M_{14} it creates a Response checkpoint. Here the Recovery line is confined to processes P_1 and P_2 only.

Rollback scheme

In order to clearly specify the operations of the Rollback scheme the following terms are defined.

(1) $RM^{qr}_{xjyk}(t)$ (Received message set): Set of messages sent out by the process P_q before its checkpoint X^q_{xj} which are received by process P_r after its checkpoint X^r_{yk} as observed at time "t".

(2) $PM^{qr}_{xjyk}(t)$ (Pending message set): Set of messages sent by P_q before its checkpoint X^q_{xj} which have not been received by process P_r even after its checkpoint X^r_{yk} as observed at time t.

(3) $VM^{qr}_{xj}(t)$ (Valid message set): Set of "unrevoked" messages sent by P_q to P_r after P_q has crossed its checkpoint X^q_{xj} as observed at time t.

Consider two processes P_1 and P_2 in a n process system. Due to errors detected let a simultaneous rollback to the set S:(RL_{x1}, RL_{y2}, RL_{z3}) be in effect. Let recovery lines and the interactions between the processes be as shown in fig.B.2.

Let P_1 receive the Rollback messages in the order $R\langle y,2\rangle$, $R\langle z,3\rangle$ and $R\langle x,1\rangle$. Let these messages be received by P_1 at t_1, t_2 and t_3 respectively. Let P_2 also receive the Rollback messages in the same order but at t_4, t_5, t_6 respectively. Let $t_4 > t_3$.

Assume that for simplicity a message M gets discarded from the channel only when it is scanned and its RCP depends on atleast one element in the ARM list say $R\langle j,a\rangle$ of the channel. We denote this as $M > R\langle j,a\rangle$, the complementary condition is denoted by $M < ARM$ i.e.

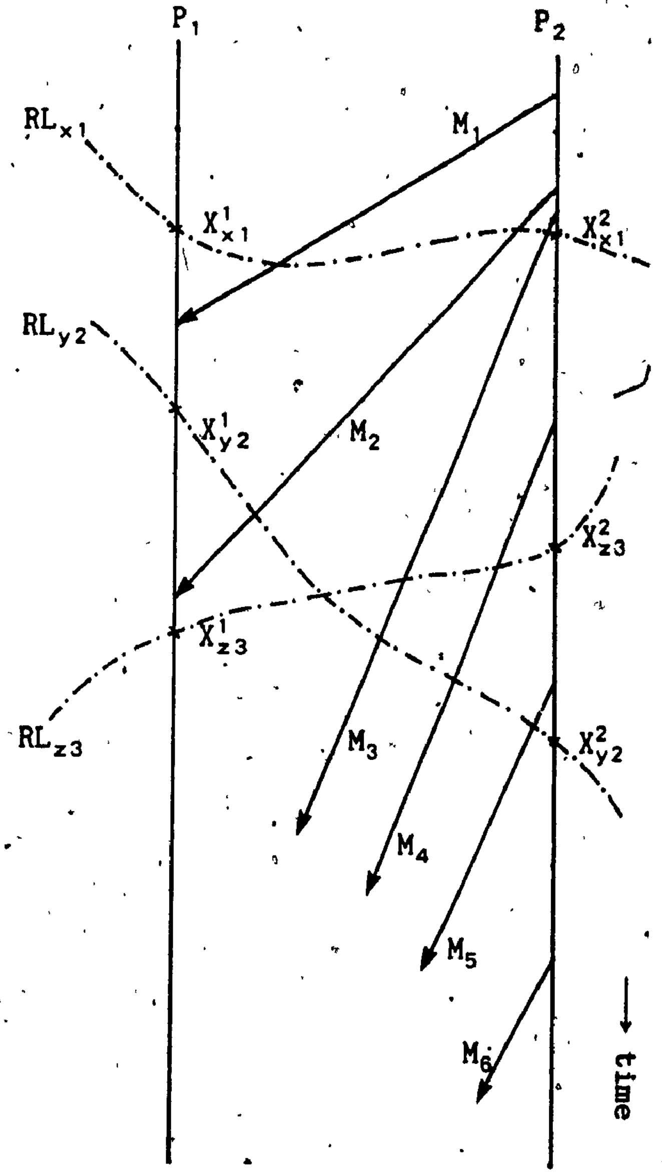


Figure B.2: Illustration of the Rollback Scheme

M does not depend on any element in the ARM list of the channel.

Possible sequence of events that might occur during recovery for this example.

At time t_1: P_1 receives $R\langle y,2\rangle$ and completes rollback to X^1_{y2} by $(t1+\delta t)$

$$RM^{21}_{x1x1}(t_1) = (M_1, M_2);\ PM^{21}_{x1x1}(t_1) = (M_3)$$

$$Channel_{21} = M_3\ M_4\ M_5\ M_6$$

At time $t_1+\delta t$

$$RM^{21}_{x1x1}(t_1+\delta t) = (M_1);\ PM^{21}_{x1x1}(t_1+\delta t) = (M_2, M_8)$$

$$Channel_{21} = M_2\ M_3\ M_4\ M_5\ M_6$$

$$ARM_{21} = R\langle y,2\rangle$$

Similar actions occur at t_2 and t_3 on receiving $R\langle z,3\rangle$ and $R\langle x,1\rangle$.

At time $t_3+\delta t$

$$Channel_{21} = M_1\ M_2\ M_3\ M_4\ M_5\ M_6$$

$$ARM_{21} = R\langle y,2\rangle\ R\langle z,3\rangle\ R\langle x,1\rangle$$

Assume that partial re-executions occur between rollbacks of process P_2. Then a possible state of the channel if we assume that P_1 is frozen till $t = t_6 + \delta t$.

$$Channel_{21} = M_1M_2M_3M_4M_5M_6R\langle y,2\rangle \mid M_6R\langle z,3\rangle \mid M_5M_6R\langle x,1\rangle \mid M_4M_5M_6$$

after t_4 after t_5 after t_6

$$ARM_{21} = R\langle y,2\rangle\ R\langle z,3\rangle\ R\langle x,1\rangle$$

Assume P_1 scans the channel sequentially

(1) Sees M_1: since $M_1 < ARM_{21}$ it is accepted

So is the case for M_2 and M_3

(2) Sees M_4: $M_4 > R\langle x,1\rangle$ it is discarded

So is the case for M_5 and M_6.

(3) Sees $R\langle y,2\rangle$: Deletes entry from ARM list. ARM_{21}: $R\langle z,3\rangle$, $R\langle x,1\rangle$

(4) Sees M_6: Since $M_6 > R\langle z,3\rangle$ it is discarded

(5) Sees $R\langle z,3\rangle$: Deletes entry from ARM list. ARM_{21}: $R\langle x,1\rangle$

(6) Sees M_5: Since $M_5 > R\langle x,1\rangle$ it is discarded.

So is the case with M_6

(7) Sees $R\langle x,1\rangle$: Deletes entry from ARM list. $ARM_{21} = \emptyset$

(8) Sees M_4: As $M_4 < ARM_{21}$ it is accepted.

So is the case for M_5 and M_6

Therefore the sequence of messages received by P_1

$$= (RM^{21}_{x1x1}(t_1),\ PM^{21}_{x1x1}(t_1),\ VM^{21}_{x1}(t>t_6))$$

= desired computation sequence