National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Attaining Transient Fault-tolerance in Distributed Systems

Basudeb Dash

A Thesis

in

The Department

of

Computer Science

Presented in Partial fulfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

May 1993

ISBN 0-315-97627-6

Canada

ABSTRACT

Attaining Transient Fault-tolerance
in Distributed Systems

Basudeb Dash, Ph. D.
Concordia University, 1993

Two alternate assumptions can be made regarding the effects of transient faults: (i) only the application system is affected by fault vs (ii) both the application system and augmentation for fault tolerance are affected by fault. Although, in much of the work on transient fault tolerance the former is assumed, the latter is considered a pragmatic way as most often the effects of transient fault cannot be contained to the application system. Transient fault-tolerance assuming the latter has been known as self-stabilization.

Safety properties of systems can be expressed in terms of the states the system is allowed to enter (safe states) and/or the state changes that are allowed to happen (safe transitions) during execution. Safe transition properties are most primitive forms of safety properties. We present general strategies for the detection of violations of safe transition properties which can be followed by rollback and recovery for transient fault-tolerance.

We present general strategies for augmenting systems with regular behaviour to make them self-stabilizing. Since self-stabilizing systems can function normally starting from any arbitrary initial state, they can get rid of initialization protocols in distributed

systems thereby enhancing their robustness. As per our strategy, self-stabilizing extensions of systems can be obtained by a preprocessor thus eliminating adhocness in designing of self-stabilizing systems. Our strategy is applicable to a large class of systems. We demonstrate the simplicity and intuitiveness of our strategy by obtaining self-stabilizing extension of token ring, alternating bit protocol and resource manager.

Acknowledgements

I gratefully acknowledge the superb guidance and extraordinary assistance of my thesis supervisor Professor H. F. Li, during the entire period of my study in Concordia University leading to this great distinction in life. Out of the many valuable things I learned from Dr. Li, the one that deserves the most recognition is "there are simple solutions to certain very difficult problems but are often hard to find. It is the simplicity that brings elegance". The experiences gained from Dr. Li will be most valuable for the future.

I sincerely thank Canadian Commonwealth Scholarship and Fellowship Plan and Government of India, for offering me a scholarship for pursuing my doctoral studies. I also thank Uttar Pradesh Agro Industrial Corporation and its management for granting me study leave. I gratefully acknowledge the encouragement and advice of Mr. R. M. Sethi.

I owe a great deal to my wife (Mana), son (Babuni) and daughter (Jhumuni) for their love and understanding. I am indebted to them for giving me comfort, happiness and most of all the reasons to fulfil my ambitions. I am grateful to my (late)father, mother, brother, brother-in-laws and sisters for their support and encouragement.

My sincere thanks to the Faculty members and the supporting staff of the Department of Computer Science for providing facilities and a conducive environment for conducting my studies and research. I would like to thank all the friends, especially Jamil, and colleagues for their help and support.

Dedicated to Our Parents

(Late) Sj. Arjun Dash

Smt. Ratnamani Dash

Sj. Sukadeb Kar

Smt. Susila Kar

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

Computing systems sometimes do fail to deliver the specified services due to a variety of reasons including design faults, implementation faults, run-time faults, malfunctioning of components etc. ([John89], [LeeP90], [Mili90]). A system is called fault-tolerant if it can continue to provide the specified services despite occurrence of faults. Fault-tolerance is a major consideration in systems and has been the focus of designers and researchers from the beginning. The faults that occur during the operation of a system can be grouped into two main categories: (1) permanent faults - faults that cause permanent failure of the components and (2) transient faults - faults that do not cause permanent failure of components but may cause certain components to function arbitrarily for a short duration. The recovery from permanent faults is often dealt with by detection of the faulty components followed by either isolation of the faulty components and reconfiguration or by repair and replacement of the faulty components ([John89], [LeeP90], [Mili90]). Failures caused by transient faults are relatively more frequent than those caused by permanent faults ([Cris91]). The most widely used approach of redundancy in software and/or hardware for fault-tolerance although increases the reliability of the systems but falls short of making them truly fault-tolerant ([John89], [Mult89], [LeeP90], [Mili90]).

There are systems that can tolerate transient faults of various types. For example,

the window protocol can tolerate faults that may cause loss, corruption, duplication and re-ordering of messages in the channel. However, the protocol will enter into livelock if the counters at the transmitter and receiver are not properly synchronized ([Goud91]).

1.1 Transient Fault-tolerance

Computing systems are often specified in terms of two types of properties, such as (i) safety properties - that specify something bad never happens and (ii) progress properties - that specify something good eventually happens. For example, at most one process can execute in the critical section at any given time (safety) and a process intending to enter the critical section should be allowed to do so within finite time (progress). Temporary malfunctioning of component(s) in a system causing undesirable behaviour (violations of safety and/or progress) is called transient fault. For example, (i) in token based systems, token may be lost or extra tokens may be created, (ii) in asynchronous message passing systems, messages may be lost, corrupted, re-ordered or duplicated in the channel, and (iii) a variable may assume an arbitrary value due to transient fault. Although transient faults do not persist, their effects may remain and continue to cause undesirable system behaviour. For example, if an extra token is created due to fault in a token based access control protocol, mutual exclusion condition may be violated on a continuous basis.

In the design of transient fault-tolerant systems, the following assumptions can be made regarding the effects of fault : (1) the application system is affected by fault but the

fault detection and recovery mechanisms are not affected or (2) the application system and the augmentation for fault detection and recovery mechanism could be affected by fault. The former is assumed in most of the work on transient fault-tolerance. However, more recently, the latter has been considered as the most pragmatic approach in transient fault tolerance. Transient fault-tolerance assuming the latter has been called by Dijkstra, and extensively studied by Gouda and others, as self-stabilization([Dijk74], [Goud91]).

Dijkstra, in [Dijk74], defined the term self-stabilization to characterize those systems that can reach a legitimate state (a state reachable from proper initial state) from any arbitrary initial state in finite time and continue to remain in legitimate states thereafter. An arbitrary state means all the state variables including program counter(s) can have arbitrary values. However, the program code is assumed to be not affected by fault. An arbitrary initial state can be caused by initialization fault or by transient fault. Ability of self-stabilizing(SS) systems to reach a legitimate state from any arbitrary state in finite number of steps and continue to remain in legitimate states thereafter has the following significant advantages :

(i) non-reliance on proper initial state

(ii) tolerant to all types of transient faults that can affect state variables and program counters

(iii) no outside intervention for fault detection and recovery

(iv) no need of any redundancy in hardware and/or software for transient fault-tolerance

and (v) no check-pointing is needed for recovery

Since SS systems are capable of delivering specified services starting from an arbitrary state, the protocols for initialization in distributed systems will be unnecessary thereby eliminating initialization overhead. Further, absence of redundancy in SS systems has significant pay off in achieving transient fault-tolerance over the commonly adopted approach of replication. Check-pointing for backward recovery often contributes significant overhead during the operation of the systems. No need to do check-pointing for recovery will be attractive in reducing operational cost in SS systems.

1.1.1 Previous Work

Redundancy in various forms (Hardware redundancy, software redundancy, time redundancy) has been the common approach for transient fault-tolerance ([John89], [LeeP90], [Mili90], [Aviz85], [Bril87], [Kope90], [Cris91]). Fault masking is one such approach to tolerating transient faults ([John89], [LeeP90], [Mili90]). Fault masking is usually achieved by means of error-correcting codes in memory and majority voting schemes (as in TMR systems - triple modular redundancy, or NMR systems - N-modular redundancy, redundant hardware modules etc [Aviz85], [Bril87]). Although redundancy provides a powerful general strategy for fault-tolerance, it comes with high cost of implementation. Another common strategy for transient fault-tolerance has been detection of fault followed by recovery ([John89], [LeeP90], [Mili90]). Detection of fault is usually accomplished by a watchdog or monitor to look for events or states that should not occur in the system. The detection of faults is usually accomplished by acceptance test on the

state of the system during execution. The acceptance tests are derived from the specification of the system. The goal of fault-detection is to look for violation of safety properties of the system. The states that should not arise in the course of the operation of the system are detected by violation of certain invariant properties. Further, application of such acceptance tests on all states during execution gives rise to exponential complexity because of concurrent events in a distributed system. Safety properties can be expressed not only in terms safe states as characterised by invariants but also by safe transitions ([Chan88]). There has not been sufficient attempt to develop detection strategies to control the exponential blow up in complexity. Moreover, detection of violation of safe transition properties has not been addressed so far.

Dijkstra ([Dijk74]) introduced the concept of self-stabilization into computer science and demonstrated the possibility of designing SS systems by means of three solutions on a token ring. Later, he proved one of his solutions to be correct in [Dijk86]. In Dijkstra's design, transitions refer to the state variables of neighbouring processes and hence are interfering. Kruijer ([Krui79]) designed a SS token system on tree-topology. Dijkstra's brilliant idea of self-stabilization as a potential strategy for transient fault-tolerance didn't get the due attention until Lamport's stunning remark ([Lamp84]). Lamport's appreciation of Dijkstra's work and its importance in fault-tolerance has received considerable attention recently. This is evident from the magnitude of published work and ongoing work in a wide spectrum of application areas (in fact recently self-stabilization has been listed in a number of international conferences as one of the primary areas of interest). At least four Ph. D. dissertations ([Brow87], [Mult89],

[Aror92]), including this one, have been devoted to issues related to self-stabilization.

The potential of self-stabilization has been demonstrated in mutual exclusion and producer-consumer problems in distributed systems. Brown ([Bown87], [Brow89]) developed SS token systems and SS solutions to drinking philosophers problem without requiring atomicity beyond single processor. Afek and Brown ([Afek89]) presented a SS solution to alternating bit protocol. However, their solution requires at least three control numbers and an infinite aperiodic sequence of numbers to be generated by the sender. In the process of designing a SS solution, they seem to have misused the name "alternating bit protocol" as in their system neither there are two control bits nor they alternate. Gouda and Multari ([Goud91], [Mult89]) developed SS connection management protocol, window protocol and a ring network reconfiguration protocol. Multari ([Mult89]) presented certain properties a protocol should possess before it can be made SS. Gouda and Multari ([Goud91]) proved the necessity of timeout actions in at least one of the processes for self-stabilization in an asynchronous message passing system. Abadir and Gouda ([Abad92]) have applied the concept of self-stabilization in synchronous digital circuits for building stabilizing computer. There has been other custom design of SS systems for applications such as unison ([Goud90]), maximal matching ([HsuS92]), dining philosophers ([Hoov92]), clock synchronization ([LuMe90]), constructing breadth-first trees ([Huan92]). Very recently, in a survey paper, Schneider ([Schn93]) has summarized the relevant work pertaining to self-stabilization.

The above mentioned SS systems are custom made keeping self-stabilization as a requirement in the design. Although the existence of such systems affirmed that self-

stabilization is a useful concept applicable to a wide variety of applications, it failed to provide a universal strategy to design a SS system for an arbitrary application. This approach for designing SS systems is adhoc and seems to be based on trial and error. In his very first work on self-stabilization [Dijk74] Dijkstra writes that " For more than a year - at least to my knowledge - it has been an open question whether nontrivial (e.g. all states legitimate is considered trivial) SS systems could exist". Dijkstra then writes in [Dijk82] that

> "*Again I beg my intrigued readers to stop reading here and try to solve the stated problem themselves, for only then will they (slowly!) build up some sympathy with my difficulties: the problem has been with me for many months, while I was oscillating between trying to find a solution - and many an at first sight plausible construction turned out to be wrong! - and trying to prove the non-existence of a solution. And all the time I had no indication in which of the two directions to aim, nor of the simplicity or complexity of the argument - if any ! - that would settle the question.*"

Katz and Perry ([Katz90]) presented another approach in designing SS systems. Their approach is to create SS extension of existing system by superimposition of SS control based on SS global state detection and SS reset. They developed a SS global state detection algorithm that records the global state of the system repeatedly and test for illegality. Arora and Gouda ([Aror90]) developed a SS distributed reset algorithm.

In the absence of any stable store for data (which is assumed in the SS context), the applicability of SS extension based on SS control is restricted to systems in which (i)

safety can be expressed as invariants and (ii) any state that satisfies the invariant is reachable from all possible initial states. The SS global state detection strategy of [Katz90] ensures that eventually some process can obtain intermittent global states of the system. There are a variety of systems, including the following, where detection of some global states of the system eventually cannot be used for the purpose of self-stabilization.

(1) Systems with the following characteristics : (i) multiple initial states and (ii) the set of states reachable from one of the initial states is disjoint from those reachable from some other.

Systems whose computation is dependent upon the initial data naturally fall into this category. For example, the result from a distributed sorting algorithm is dependent upon the set of the data items given as input to it. In such systems, it will be impossible to determine if a state that satisfies the invariant is reachable from the initial state at which the execution started. Hence, if a transient fault causes the system to reach and then execute in states that are not reachable from the initial state but otherwise satisfy the invariant, it will not be possible to detect this by SS global state recording strategy. Hence, the system may deliver services quite different than it was supposed to with the approval of the fault tolerance mechanism.

(2) Systems in which safety is specified not just by invariants but also by ***P unless Q*** or ***P stable*** (in terms of safe transitions), where P and Q are predicates on the system state.

Consider the general committee co-ordination problem which represents the class of general agreement problems in computing systems. There are different committees and each committee has a set of members. A member may sit in more than one committee. However, each member can attend at most one committee at any given time. A committee can convene when all its members are free. Hence, the safety requirements of such a system are (a) a member can attend at most one of the committees of which he/she is a member (expressed as an invariant) and (b) a committee cannot convene unless all its members are free (safe state transitions expressed by P unless Q). In such systems, there are properties not only to be satisfied by states but also by adjacent state pairs (safe transition properties) in the specification. Hence, SS global state detection strategy cannot be applicable to such systems as detection of intermittent global states of the system reliably will not be able to detect violation of safe transition properties. Therefore, the strategy based on SS global state detection cannot provide SS extension of such systems.

(3) Systems whose behaviour is specified in terms of actions instead of a set of temporal properties.

1.1.2 Overview of Our Work

Safety properties specify what should not happen during the run of the system (i.e states in which system should never be and/or transitions which should never happen during execution of the system). Hence, safety properties of systems can be expressed in terms of safe states and/or safe state transitions. The safe states are characterised by

predicates that hold in all states of the systems (expressed by invariants). Where as safe state transition properties specify what transitions are allowed from safe states. Safe transition properties can be expressed in the forms such as (i) P unless Q, (ii) Stable P, where P and Q are predicates on the state of the system ([Chan88]). Amongst all the above forms, P unless Q is the most primitive (invariant and stable can be expressed in terms of unless). We consider the detection of violations of properties of the form P unless Q to be fundamental in the detection of violation of safety that could arise due to fault. We address the detection of violations of P unless Q as the first step in transient fault-tolerance. However, for detection of violations of safe state transition properties, it is required to test for possible violations at every state change during the execution. This will give rise to exponential complexity. Our detection strategy is aimed to control the complexity of detection by recognizing the nature of predicates involved in expressing safety. Our strategy can be applied to detect violations of both safe state and safe transition properties. This can be followed by recovery (backward or forward as the case may be) for transient fault-tolerance. However, we don't address the problem of recovery in this dissertation (refer to [Venk87], [KooR87] and [Stro85] for check-pointing and rollback/recovery). Our strategy can also be used for debugging applications.

Fault-tolerance is an orthogonal property of systems for increasing their robustness and is not often a part of the original correctness requirement. Hence, issues related to fault-tolerance should be deferred until the system is designed and verified. We advocate this philosophy in our work on self-stabilization. For a large class of systems, whose behaviour can be specified by a regular expression over actions, the burden of making

them SS need not be put on the designer. Instead, such system can be made SS automatically by a preprocessor. We present a general strategy for obtaining SS extension of systems with regular behaviour. Our augmentation strategy is distributed in nature and provides the earliest possible detection of fault. Early detection of faults helps to control the spread of fault and subsequent damage caused by it. Our strategy, in general, uses vector clock, choice clock and version numbers for fault detection and recovery. The vector clock is used to monitor causal relationship between system events. Choice clock is used to monitor compatibility among choices and version numbers are used to enforce stabilizing reset should the need be. We customise our strategy by employing forward recovery in systems with deterministic behaviour. Besides our strategies being general, they also provide a very intuitive and simple understanding of the steps in obtaining SS extension of systems. We feel the later part is lacking in most of the work on SS. We believe, it is important to explain the strategy in obtaining SS extension of an existing system than a custom made SS solution that provides no insight to others. It is interesting to observe that some of the SS solutions use logical counters in their design ([Dijk74], [Goud90]) without mentioning their relevance in SS. Our strategy is bound to help the user in obtaining a SS solution to existing systems.

We demonstrate our strategy by creating SS extensions of (i) token ring (ii) alternating-bit protocol and (iii) resource manager. The SS solutions obtained through extension for token ring and alternating bit protocol are easier to understand than the solutions already proposed. We also present SS extension of one more system, resource manager, broadening the scope of self-stabilization.

1.2 Organization

In this dissertation, we address issues related to transient fault-tolerance and self-stabilization in distributed systems.

In chapter 2, we address the problem of detection of safety violations in distributed systems and present efficient algorithms. Our strategy is meant for detection of the most primitive form of safety properties (**unless**). We explore the simultaneity of intervals to facilitate efficient detection without suffering the ill effects of state explosion in case of predicates consisting of locally detectable components.

In chapter 3, we characterise systems that can be extended to become SS. We define a set of production rules. If the behaviour of a system can be deduced by these sets of production rules, it can be made SS by our strategy. We also define self-stabilization and SS extension.

In chapter 4, we present a general strategy for SS extension. First of all, we develop the strategy for fault detection. The detection strategy is distributed and is guaranteed to detect fault. Then we present a general strategy for recovery which can be triggered in case fault is found.

In chapter 5, we present a general strategy for SS extension of systems with deterministic behaviour using forward recovery strategy.

In chapter 6, we demonstrate our strategy by creating SS extensions of three systems such as (i) token ring (ii) alternating-bit protocol and (iii) resource manager. We present the original system and then augment them to obtain the self-stabilizing extension.

Finally, conclusion and future research directions are discussed in chapter 7.

CHAPTER 2

DETECTION OF SAFETY VIOLATIONS

In this chapter, we consider the detection of violations of safety properties of the form **P unless Q**, where P and Q are distributed predicates on the program state. We develop an interesting relationship between simultaneity of local predicates and P unless Q in distributed systems. We present an algorithm for detection of violations of P unless Q that effectively avoids state explosion problem. The worst case complexity is quadratic in the number of predicates in P and Q. The detection of violations of safety properties can be used for transient fault-tolerance and distributed debugging. We also present a strategy to efficiently reduce the storage requirement of the algorithm.

2.1 Introduction

In the specification and verification of programs, two kinds of properties are of primary importance - ***safety properties*** and ***progress properties***. Safety properties assert that something bad never happens where as progress properties assert that something good eventually happens. The safety properties of a program can be expressed in the following forms: ([Chan88]) (1) ***P unless Q*** or (2) ***stable P (P is stable)*** or (3) ***invariant P (P is invariant)***, where **P, Q** are predicates on program states. Amongst all the above

mentioned forms of safety properties, ***unless*** is the most primitive (***stable*** and ***invariant***, are special cases of ***unless***). We establish interesting relationships between P unless Q and the concept of simultaneity of local predicates in distributed system. By doing this, we are able to develop an efficient detection strategy for checking simultaneity and hence safety conformance checking by effectively avoiding the state explosion problem.

We consider safety specifications of the form P unless Q. However, our definition of P unless Q, as given below, differs slightly from that of Unity ([Chan88]). When an execution fails to satisfy this requirement specification, due to operational failure, a monitor superposed on the application can detect such violations. For transient fault tolerance, recovery can be initiated after detection of safety violation and the system could restart from a safe (legitimate) state. This safety conformance testing can also be used to detect design faults (debugging applications). In case of testing/debugging a design, safety conformance checking can be used to assert when and where (in the state space) violations of safety has occurred. The safety conformance checking of general type has not been addressed so far which we feel is one of the most important aspects in distributed debugging ([Lebl87], [Mill88], [McDo89], [Wald91]).

The state of a system consists of values of all the state variables distributed at different processes including program counters and the contents of the channels. As per our definition of P unless Q, over all the states of a computation E, there should never be a state C at which P holds and $\neg(P \wedge Q)$ holds at C' where C' is a next state (state

reached by executing a single atomic action at some process) of C. We find this definition for the purpose of specifying safety properties more useful than the one given by Chandy ([Chan88]). Since, the qualification is over all the states of the computation and the number of states in a computation is exponential to the number of events in E (in the worst case), in general, safety conformance testing is prohibitively expensive. However, this exponential cost could be effectively removed. In this thesis, we present a scheme whereby the cost of the detection of violations is reduced to quadratic in the number of processes and linear in the number of events.

We also establish relationship between simultaneity (Lamport's "can causally affect" relationship between intervals[Lamp86b]) of local predicates and P unless Q. We recognize that within a (logical) time window among processes, a distributed predicate may hold in some state iff there exists a set of simultaneous intervals containing the respective local predicates of the distributed predicate with the desired values. Violations of P unless Q caused by the occurrence of some event e can now be checked by restricting attention to a time window defined by e within which appropriate simultaneity relationship can be checked.

2.2 System Model

An *asynchronous distributed system (program)*, which we say *distributed system* or *distributed program* henceforth, consists of several processes without any common

memory. The processes run at unpredictable but non zero speed and communicate by messages. Messages experience unpredictable but non zero communication delay. The processes are strongly connected with each other without necessarily having direct communication link (channel) between every pair of processes. Each process is a sequence of events where each event is an atomic transition of the local state that takes no time (i.e. execution of an *atomic step*) for execution. There are three different types of events at any process : *internal event*, *receive event*, and *send event*. An *internal event* causes a change of state. A *receive event* causes receipt of a message from one of its incoming channels and the local state to be updated by the value of the message. A *send event* causes a message to be sent on one of its outgoing channels.

We consider partial order model of execution of a distributed program. The events in the system are partially ordered ([Prat87]). A distributed computation is represented by (E, $\prec$), where E is a set of events and $\prec$ is an irreflexive partial order on E, (***causality relation***). For a given computation, $\forall e, e' \in E$, $e \prec e'$ holds if one the following holds:

(a) e and e' are events of the same process and e occurs before e'.

(b) e is the send event of a message and e' is the corresponding receive event.

(c) $\exists\, e'' \in E$ such that $e \prec e''$ and $e'' \prec e'$.

The events at a process are totally ordered. The events from different processes are ordered by the *causality* relationship ([Lamp78]). The *global state* of such a

distributed system consists of local states of all the processes and channels ([Chan85]). There is no way a process can know the instantaneous global state of the system. This has led researchers and practitioners to consider *global snapshots (a global state that could have occurred if processes would have taken snapshots simultaneously)*, as defined in [4], for specification, design and analysis of distributed systems. Execution of an event at a process changes the global state of the system. Due to inherent asynchrony amongst processes, concurrent events at different processes may cause an omnipresent observer (an idealized external observer that can see instantaneous global state of the system) to experience different sequences of global states in different runs of a distributed system even in the absence of non-determinism.

2.3 Relevant Notions and Definitions

A predicate is boolean valued function defined on the state of a program. A predicate that is defined on the local state of a process is called a **local predicate**. Unless otherwise mentioned, a **local predicate** will be called as **predicate** henceforth. Hence, a predicate can be evaluated at the process on whose local state it is defined and its value can be changed atomically by an event at the process. Since a predicate is defined on the state of a process, we say that each predicate is **owned** by some process. A **distributed predicate** is defined as a logical expression on local predicates with the usual logical operators **AND**, **OR** and **NOT**. A distributed predicate can be expressed equivalently in conjunctive normal form where each component is a disjunction of local predicates. Since,

no process can obtain the instantaneous global state of the system, a distributed predicate can not be evaluated at any process instantaneously.

Definition : A consistent cut C of a set of events E is a finite subset $C \subseteq E$ such that $\forall e \in C$ and $e' \prec e$ implies that $e' \in C$.

Hence, a consistent cut defines a prefix of the computation. From the assumption of the events of a process being totally ordered, the events in C can be labelled as e_{ij}, where e_{ij} denotes j^{th} event of process P_i. Thus, a consistent cut can be denoted by $(e_{1k1},...,e_{nkn})$, where e_{iki} is the last event of P_i contained in C.

Definition : The **state cut**, C_s of a consistent cut C is the set of last events of each process contained in the cut C. (i.e. $C_s = \{ e_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq ki$ such that $\forall e_{im}$ of $P_i \in C, e_{im} \prec e_{ij}\}$) Hence, a consistent cut also represents a state of the system.

Definition : The **global state** (**state** in short) of a system associated with a consistent cut C is the collection of local states of the processes immediately after the occurrence of the events in state cut of C and the set of messages sent but not yet received.

We use C to denote a consistent cut and also the state of the system associated with C wherever the context is clear. The set of consistent cuts of E represents all the states of the system during an execution. Hence, the set of all consistent cuts of a given

computation gives rise to the state-space of the computation with transition between states given by the occurrence of an event.

Definition : An event e is said to be enabled at C, iff $e' \prec e$ implies $e' \in C$ and $e \notin C$.

This means that the event e could be executed next by some process at state C yielding the next state.

Definition : For a given state C, the states that can be reached by executing an enabled action at C, is given by $N(C) = \{ C' \mid C' = C \cup \{e\} \}$, where e in enabled at C.

Definition : An **e-transition** is the occurrence of the event e in some state C yielding a new state $C' \in N(C)$.

An event e enabled at C could cause an e-transition by changing the state.

Definition : For an event e, ***MIN(e)*** is the set of events in E that causally precede e (causal prefix leading to the event e) is given by $MIN(e) = \{e' \mid e' \prec e\}$ and ***MAX(e)*** consisting of all the events that are not causally preceded by e (maximal prefix of e without including e) is given by $MAX(e) = \{e' \mid \neg(e \prec e')\}$.

MIN(e) and ***MAX(e)*** corresponds to the minimal and maximal state, respectively,

at which e is enabled.

Definition : Events e and e' are said to be concurrent, denoted by e // e', iff $\neg(e \prec e') \wedge \neg(e' \prec e)$.

The set of events that are concurrent with e, denoted by **CON(e)**, is given by CON(e) = { e' | e // e'}.

MIN(e), MAX(e) and CON(e) are shown in Fig. 2.1.

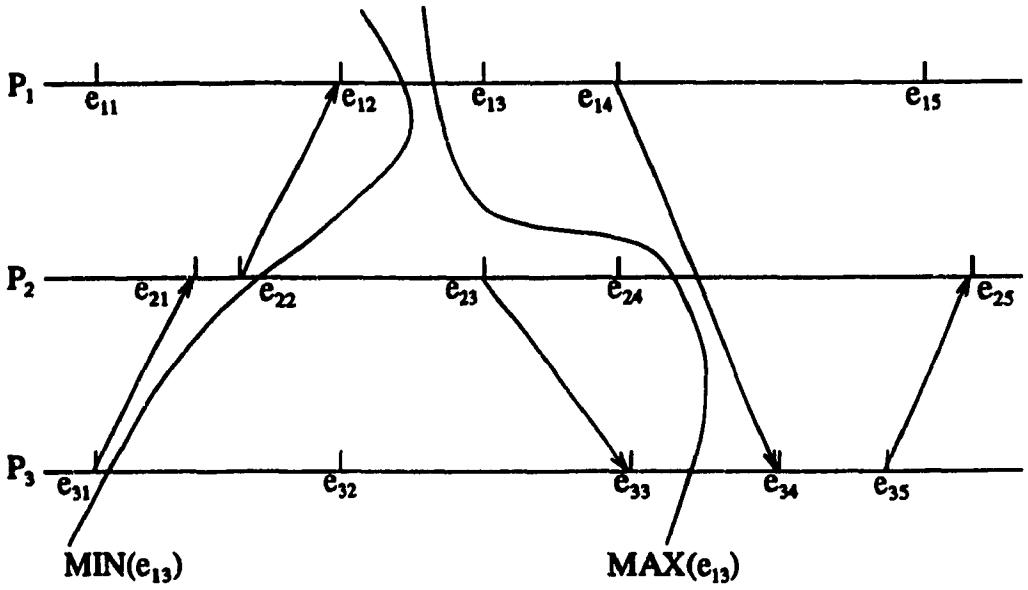


Fig. 2.1 $MIN(e_{13}) = \{e_{11}, e_{12}, e_{21}, e_{22}, e_{31}\}$,

$CON(e_{13}) = \{e_{23}, e_{24}, e_{32}, e_{33}\}$, $MAX(e_{13}) = MIN(e_{13}) \cup CON(e_{13})$

Lemma - 2.1 : For an event e, MAX(e) consists of the events that casually precede e or concurrent with e i.e. $MAX(e) = MIN(e) \cup CON(e)$.

Proof : As per the definition, MAX(e) consists of all the events that are not preceded by e i.e. MAX(e) = {e'|¬(e ≺ e')}. So, MAX(e) = {e'|(e' ≺ e) ∨ (e'//e)}. As per definition, MIN(e) = {e'|e'≺e} and CON(e) = {e'|e'//e}. Hence, MAX(e) = MIN(e) ∪ CON(e).■

Lemma - 2.2 : An event e is enabled at C iff MIN(e) ⊆ C ⊆ MAX(e).

Proof : (⇒) As per the definition, MIN(e) contains all the events that causally precede e. Hence, MIN(e) is the minimal state at which e is enabled. Assume that e is enabled at C. Hence, C does not contain e. Let us assume that C ⊂ MIN(e). If C ⊂ MIN(e) and e is enabled at C, then ∃ e' ≺ e such that e' ∉ C. Thus the event e cannot be enabled at C contradicting the assumption. Hence, MIN(e) ⊆ C. We are left to prove that if e is enabled at C then C ⊆ MAX(e). Assume that e is enabled at C and C ⊃ MAX(e). MAX(e) is the maximal state at which e is enabled. So, MAX(e) should contain all the events in C i.e. MAX(e) ⊇ C which contradicts C ⊃ MAX(e).

(⇐) MIN(e) ⊆ C ⊆ MAX(e) implies that C contains all the events that causally precede e but does not contain e. As per the definition, e is enabled at C. ■

Hence, an event e is enabled at every state between MIN(e) and MAX(e). A predicate is ***stable*** if it remains *true* once it becomes *true*. In general, a predicate may change from true(false) to false(true) several times during the execution. For a given predicate, during an execution, some event may turn it from false(true) to true(false) and it may remain false(true) until another event toggles it. Hence, a predicate remains stable

in intervals as per the local time of a process, in an execution. Thus an interval can be associated with each local predicate during which it is false(true). We call them as stable intervals of the predicate.

An interval along a process line during which predicate a is stable is denoted by $I_a = (e_{as},...,e_{af})$, such that e_{as} (start event) changes a to true/false and e_{af} (finish event) changes to false/true and the events in between e_{as} and e_{af} do not affect a. The process that owns the predicate a can maintain the value of a to be true/false during the intervals bounded by the times after the occurrence of e_{as} and up to but not including the occurrence of e_{af}. We denote I_a as the interval in which a is true. Let intervals I_a and I_b be denoted by $(e_{as},...,e_{af})$ and $(e_{bs},...,e_{bf})$ respectively.

Definition : Two intervals I_a and I_b are said to be ***simultaneous*** iff $\neg(e_{af} \prec e_{bs}) \wedge \neg(e_{bf} \prec e_{as})$. A set of intervals are simultaneous iff every pairs of intervals in the set are simultaneous.

2.4 Problem Statement

We consider properties of the form ***P unless Q***, where **P** and **Q** are unstable distributed predicates. For a predicate P, P holds at state C is denoted by P(C). The definition of ***P unless Q***, in our model of execution, is given as follows :

Definition : Given distributed predicates P and Q, P unless Q is satisfied by an execution E iff

$$\forall C \subseteq E, C' \in N(C) \subseteq E, P(C) \Rightarrow P(C') \vee Q(C') \qquad (r1)$$

This means that if P holds at some state C, then (P ∨ Q) holds at all possible next states of C. In this work, we address the problem of detecting any violations of ***P unless Q*** during an execution.

2.5 Detection of Violation

As per the definition, an execution E satisfies P unless Q iff there does not exist an event e enabled at $C \subseteq E$ such that P(C) and $\neg P(C') \wedge \neg Q(C')$ are both true, where C' is the state reached by executing e at C.

Definition : An **e-transition violation** of P unless Q is said to have occurred iff an e-transition violates (r1) above.

Lemma - 2.3 : An e-transition violation cannot be attributed to an event e, if over all e-transitions,

(a) e does not alter any predicate of P.

OR (b) e changes P from false to true.

OR (c) e changes Q from false to true.

Proof : Direct from the definition of e-transition violation. ■

Notations : Predicates P and Q can be expressed as conjunction of maxterms as follows:

$$P = \wedge_k (M_k)$$

$$M_k = \vee_j (a_{kj})$$

$$Q = \wedge_k (M_k')$$

$$M_k' = \vee_j (b_{kj})$$

Where, M_k and M_k' are k^{th} maxterms of P and Q respectively, a_{kj} is the j^{th} local predicate of M_k and b_{kj} is the j^{th} local predicate of M_k'.

The occurrence of an event e at some process P_i, can have the following possible cases of consequences that are locally decidable at P_i (i.e. P_i can conclude by observing the effects of e on its local state).

(1) e changes some predicate a_{ij} of P from true to false but none of the predicates of Q are changed.

(2) e changes some predicate a_{ij} of P from true to false and some predicate b_{rs} of Q from true to false.

(3) e does not change any of the predicates of P or Q to false.

(4) e changes P from true to false and Q is false (either e changes Q from true to false or Q is false before execution of e and e has no effect on Q) after execution of e.

As per Lemma-2.3, it can be immediately concluded that case (3) cannot lead to e-transition violation. Case (4) leads to trivial violation of P unless Q by e which is

locally decidable. Hence, we are left with identifying under what situations cases (1) and (2) will lead to e-transition violation. The possible e-transition violation scenarios are addressed in the following theorem.

Theorem - 2.1 : The occurrence of an event e leads to e-transition violation iff

for case (1) : $\exists$ C, MIN(e) $\subseteq$ C $\subseteq$ MAX(e) such that

$$P/\wedge_{k\neq i}M_k(C) \wedge \forall_{m\neq j} \neg a_{im}(C) \wedge \neg Q(C) \text{ holds}$$

for case (2) : $\exists$ C, MIN(e) $\subseteq$ C $\subseteq$ MAX(e) such that

(i) $P/\wedge_{k\neq i}M_k(C) \wedge \forall_{m\neq j} \neg a_{im}(C) \wedge \neg Q(C)$ holds

OR

(ii) $P/\wedge_{k\neq i}M_k(C) \wedge \forall_{m\neq j} \neg a_{im}(C) \wedge \forall_{t\neq s} \neg b_{rt}(C)$ holds

(X/Y denotes the value of X assuming Y to be true, M_i denotes the i^{th} maxterm whose j^{th} predicate is a_{ij})

Proof : ($\Rightarrow$) As per Lemma-2.2, an event e is enabled at C iff MIN(e) $\subseteq$ C $\subseteq$ MAX(e). So, all e-transitions could occur on states bounded by MIN(e) and MAX(e). If there is an e-transition violation at C leading to state C', there are only two possibilities

(a) $P(C) \wedge \neg Q(C)$ and $\neg P(C') \wedge \neg Q(C')$ i.e. $(P \wedge \neg Q)$ holds at state C and execution of e leads to a state C' such that $\neg(P \vee Q)$ holds at C'. This covers case (1) and (i) of case (2) above.

(b) $P(C) \wedge Q(C)$ and $\neg P(C') \wedge \neg Q(C')$ i.e. $(P \wedge Q)$ holds at C and execution of e leads to a state C' such that $\neg(P \vee Q)$ holds at C'. This covers

(ii) of case (2) above :

($\Leftarrow$) case (1) : Say, $P(C) \wedge \neg Q(C)$ holds at some state C, where $MIN(e) \subseteq C \subseteq MAX(e)$. Let $C' = C \cup \{e\}$. Thus, execution of e gives rise to $\neg P(C') \wedge \neg Q(C')$ which implies e-transition violation.

case (2) : (i) Say, $P(C) \wedge \neg Q(C)$ holds at some state C, where $MIN(e) \subseteq C \subseteq MAX(e)$. Let $C' = C \cup \{e\}$. Thus, execution of e gives rise to $\neg P(C') \wedge \neg Q(C')$ which implies e-transition violation.

(ii) Say, $P(C) \wedge Q(C)$ holds at some state C, where $MIN(e) \subseteq C \subseteq MAX(e)$. Let $C' = C \cup \{e\}$. Thus, execution of e gives rise to $\neg P(C') \wedge \neg Q(C')$ which implies e-transition violation. ■

In order to detect any violations of P unless Q, it is sufficient to check that there is no e-transition violations during the execution. As established in the above theorem, detection of violations of P unless Q need be triggered only when a process changes a component of P from true to false. This eliminates unnecessary work for detection of violations of P unless Q. We have also established that e-transition violations could only occur at states between MIN(e) and MAX(e). Hence, for detecting e-transition violations, it is sufficient to consider the states between MIN(e) and MAX(e) only to test for conditions as established in Theorem-1. There are two possible monitoring strategies : centralised vs distributed. It is believed, by many, that a distributed algorithm has certain advantages over a centralised one with respect to distribution of monitoring overhead and network congestion. We now develop a distributed algorithm for detection of violations

of P unless Q. The algorithm is initiated at a process that changes a predicate of P from true to false.

Detection Algorithm : We assume that for every event e, MAX(e) is finite. In case, MAX(e) is infinite, it is not possible to bound the search space for detecting violations of P unless Q. For the purpose of detection, we assume the existence of a time-stamp mechanism to be associated with events that is isomorphic to the event structure. Vector clock [Matt89] is one such mechanism which is assumed to be associated with events and appended to messages. Each process P_i that owns a predicate of P or Q is augmented with the following code, called a ***monitor***, for detection purpose. The monitor at a process has access to the local state of the process and can send messages to other monitors by appending its messages to the application messages. Let us say that e_{ii} and e_{ij} are the events of the process P_i corresponding to MIN(e) and MAX(e) respectively of some event e at a process P_j. The monitor assigns vector clock to each event of the process and appends vector clock of the send event with application messages.

Monitor : Upon execution of an event e by P_i, (written in guarded command form)

[] e changes some predicate a to false →

record it as the finish event of the previous stable interval and start event of the next stable interval for a. If the predicate a is a component of P, then broadcast to all processes (directly or indirectly through processes) a sampling request time-stamped with the vector clock associated with the event e and wait for responses.

[] e changes some predicate a to true →

record it as the finish event of the previous stable interval and as the start event of the next stable interval.

[] Upon receiving a sampling request from P_j →

send the recorded start and finish event time stamps of all stable intervals of predicates within the interval e_{ii} and e_{ij} to P_j. (e_{ii}, the last event of P_i that precedes e, is explicitly identified by the time stamp of e passed along with the request message while e_{ij} is identified by P_i as soon as P_i perceives the event e of P_j through application messages)

Upon receiving responses from all the processes corresponding to a sampling triggered by some event e, P_i proceeds to check if conditions as specified in case (1) or case (2) above is satisfied. If yes, then an e-transition violation is detected and appropriate action is taken depending upon the application.

We have established that for checking e-transition violations of P unless Q, it is required to check all states between MIN(e) and MAX(e). The set of states between MIN(e) and MAX(e) is exponential in the number of events between MIN(e) and MAX(e). Although, our exploration of the problem has reduced the search to the states between MIN(e) and MAX(e), we explore further to reduce the work significantly by developing and utilizing the concept of simultaneity between intervals in the detection

algorithm. The decision problem is formalized as follows :

Lemma - 2.4 : Two intervals $I_a = (e_{as}, ...,e_{ak}, e_{af})$ and $I_b = (e_{bs}, ..., e_{rt}, e_{bf})$ are simultaneous iff there exists a state cut including some event from I_a and some event from I_b such that neither of these events is the finish event of I_a or I_b.

Proof : ($\Rightarrow$) From the definition of simultaneous intervals, If I_a and I_b are simultaneous, then either e_{ak} and e_{bs} or e_{rt} and e_{as} are members of some state cut.

($\Leftarrow$) Let us assume that there exists a state cut including the events from the intervals I_a and I_b, excluding the finish events, but I_a and I_b are not simultaneous. If I_a and I_b are not simultaneous, then either $e_{af} \prec e_{bs}$ or $e_{bf} \prec e_{as}$. Since, the events in an interval are totally ordered, this implies that either all events of I_a precede each event in I_b or vice versa. Consequently, one could never find a state cut including events from I_a and I_b (excluding the finish events). This contradicts the assumption. ■

The decision algorithm utilizes the above property of simultaneous intervals of predicates to detect violations of P unless Q. Given a set of intervals of predicates, the decision algorithm looks for simultaneity between the intervals of predicates to find out whether a state exists to cause e-transition violation. The algorithm utilizes the vector clocks associated with the start and the finish events of the intervals to establish simultaneity.

At each process, the intervals of a given local predicate are totally ordered. Let, the number of intervals to be analyzed be bounded by K and number of local predicates in P and Q be m (a_1,...,a_m).

Data Structure : For each predicate, the intervals are stored in a linked list. At each node, the vector clocks of the start and the finish events of an interval are stored. We can view intervals of some predicate as a linear tree with the intervals being nodes and the precedence relationship between intervals represented by edges. In order to find out simultaneous intervals of predicates, we have to establish that there exists a set of nodes, one from each tree, such that there does not exist an edge between any pair of nodes in the set. Say, L_i is the link list of the intervals of a_i.

The decision algorithm is given as follows.

1. set i to 1.
2. Check the root node of L_i with the root nodes of L_{i+1},...,L_m.
3. If it precedes any of the root nodes
4. then remove it from the list;
5. if the list is not empty
6. then go to 1
7. else declare FAILURE and STOP
8. else increment i by 1
9. if i < m then go to 2
10. else declare SUCCESS and STOP.

It is trivial that the algorithm terminates either by declaring success or declaring failure.

Lemma - 2.5 : The decision algorithm terminates with success iff the intervals of the predicates are simultaneous.

Proof : ($\Rightarrow$) The algorithm terminates with success, when it finds that no pair of the root nodes at the list of predicates precede each other as per the description of the algorithm. This implies that the intervals represented by the root nodes are simultaneous.

($\Leftarrow$) According to definition, a set intervals are simultaneous iff there does not exist precedence relationship between any pairs of them. The first set of intervals that are simultaneous will appear at the root nodes of the lists. Hence, the algorithm will execute step-10 and terminate with success. ■

It is easy to see that the worst case complexity of the decision procedure is $O(m^2K)$.

Theorem - 2.2 : The detection algorithm, as discussed, detects e-transition violations of P unless Q, if any.

Proof : Proof follows directly from theorem-2.1 & Lemma-2.4 . ■

For detecting e-transition violations, the monitor at a process has to store all the stable intervals of predicates so that it can respond to requests from other monitors. This

is obviously too expensive in terms of storage. Hence, we develop some strategy to discard intervals at some point in time such that they will not be needed by any process thereafter. Intuitively, if a monitor can learn that no other monitor will require an interval in future, then the interval can be safely discarded.

Definition : For any state cut $C_s = (e_{1k1},...,e_{nkn})$ of a consistent cut C, an interval $I_a = (e_{as},e_{af})$ with $e_{as}, e_{af} \in C$ is **obsolete** iff $\forall i, e_{af} \prec e_{iki}$.

Lemma - 2.6 : An interval $I_a = (e_{as},...,e_{af})$ can be associated with an e-transition violation (as per the detection algorithm) only if $\neg(e_{af} \prec e)$.

Proof : As per Lemma-2.2, an e-transition violation can occur at states bounded by MIN(e) and MAX(e). For an interval $I_a = (e_{as},...,e_{af})$, if $e_{af} \prec e$ then all the events of I_a are contained in MIN(e) and hence any state at which e-transition violation may occur cannot contain any event from I_a. Therefore, I_a will not contribute to e-transition violation. ■

Lemma - 2.7 : If an interval is obsolete for C, then $\forall C' \supseteq C$, no event $e \in C' - C$ can have e-transition violation detected involving the interval.

Proof : As per definition, an interval $I_a = (e_{as}, ..., e_{af})$ is obsolete for a state cut $C=(e_{1k1},...,e_{nkn})$ iff $\forall i, e_{af} \prec e_{iki}$. If a state cut $C'= e_{ak1}',...,e_{nkn}') \supseteq C = (e_{1k1}, ..., e_{nkn})$, then $e \in C' - C, \exists e_{iki} \prec e$. As per Lemma-2.2, if the last event of an interval precedes an event

e then it can not contribute to e-transition violation. Hence, if interval I_a is obsolete for C, and $C' \supseteq C$ then $\forall e \in C' - C, e_{af} \prec e$. Therefore, I_a can not contribute to e-transition violation of any event $e \in C' - C$. ■

From Lemma-2.6, it implies that if the last event of an interval precedes some event in every process, then the interval cannot contribute the e-transition violations of P unless Q. From Lemma-2.7, once an interval is found to be obsolete then it will be obsolete thereafter. We develop an algorithm for detecting obsolete intervals as follows:

For discarding an obsolete interval, a monitor must know that the last event of the interval is contained in MIN(e) of some event in every process. This can be trivially done, provided the monitor knows the vector clocks of events of every process in the system. But this requires a (n x n) matrix to be maintained at each process and to accompany application messages. This is again expensive. We present the following modified time-stamp mechanism to decide when an interval will be obsolete. Each event is associated with vector clock as before. Each application message carries the time-stamp of the send event. We propose a voting scheme to gather common knowledge about perception of processes. A process maintains another time-stamp vector called **HTS (hand-shake time stamp)** that has N components, one for each process in the system. Each component has three fields.

Say, the i^{th} component of HTS of an event e_{jk} at process P_j is denoted by (e_{ki1}, e_{ki2}, y_{ki}). The event e_{ki1} is the latest event of P_i that has been perceived by every process in the

system at e_{jk}. The event e_{ki2} is the latest event after e_{ki1} that has been perceived by P_j at e_{jk}. The y_{ki} contains a n-bit vector. If m^{th} bit in y_{ki} is set it denotes that process m has perceived the event e_{ki2} of P_i. Hence, looking at the bits of y_{ki}, it can be known that how many processes have perceived the event e_{ki2} of P_i. Each application message also carries this vector. We now give the algorithm to maintain the HTS.

Upon receiving HTS, the monitor process at P_j does the following:

For every component of the HTS vector :

(1) If the event that appears in the second field of the component (this information is available in the vector clock) has not been perceived yet, then update the second field by the latest event that has been perceived and set j^{th} bit and reset all other bits in the third field else set the j^{th} bit of the third field.

(2) If all the bits in third field are set, then

(i) update the first field by the event that appears in the second field

(ii) set the latest event it has perceived from that process in the second field

(iii) set the j^{th} bit in the third field.

It is easy to see that the first field of any component of HTS reveals the common knowledge of all the processes.

The algorithm to discard an interval from its local store is as follows:

If there exists an interval $I_a = (e_{as}, ..., e_{af})$ with P_i such that $eaf \prec e_{kil}$ of some HTS of e_{ik}, then delete I_a from the store.

Theorem - 2.3 : Once an interval is discarded by the above algorithm, no sampling request will be received for that interval in future.

Proof : Follows directly from Lemma-2.6 and Lemma-2.7. ■

As shown above, the HTS scheme accomplishes the objective with a (n x 3) clock matrix in stead of (n x n).

2.6 Example

Consider a token ring system consisting of three processes P_1, P_2 and P_3. Let the ring to be oriented as $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$. Each process is an infinite loop given as follows.

```
LOOP
    Receive(TOKEN);
    Use(TOKEN);
    Send(TOKEN);
END
```

Let us say, the predicate c_i means P_i is in the critical section, for i = 1, 2 and 3. Safety properties for the above system can be expressed as follows:

p_1: $(\neg c_1 \wedge \neg c_2 \wedge \neg c_3)$ unless $(c_1 \vee c_2 \vee c_3) \wedge$
$(\neg c_1 \vee c_2 \vee c_3) \wedge (c_1 \vee \neg c_2 \vee c_3) \wedge$
$(c_1 \vee c_2 \vee \neg c_3) \wedge (\neg c_1 \vee \neg c_2 \vee \neg c_3)$

p_2: $(c_1 \wedge \neg c_2 \wedge \neg c_3)$ unless $(\neg c_1 \wedge c_2 \wedge \neg c_3)$

p_3: $(\neg c_1 \wedge c_2 \wedge \neg c_3)$ unless $(\neg c_1 \wedge \neg c_2 \wedge c_3)$

p_4: $(\neg c_1 \wedge \neg c_2 \wedge c_3)$ unless $(c_1 \wedge \neg c_2 \wedge \neg c_3)$

Let us say that some transient fault has caused an additional TOKEN at P_3 when P_1 is enabled with the legitimate TOKEN. Such an action causes $\neg c_3$ to change to false arbitrarily. Since P_1 is enabled with the legitimate TOKEN, the system will violate mutual exclusion as specified in the safety property (p_4). This will continue for ever. Consider the computation as shown in Fig. 2.2. The event e_{12} causes $\neg c_1$ to change to false. Due to this, a component of the left hand side of unless in (p_4) is changed to false. The monitor can compile and detect a state between MIN(e_{12}) and MAX(e_{12}) to find out a state $C = (e_{11}, e_{22}, e_{33})$, where $(\neg c_1 \wedge \neg c_2 \wedge c_3)$ is true and the execution of e_{12} at C will lead to a state where both $(\neg c_1 \wedge \neg c_2 \wedge c_3)$ and $(c_1 \wedge \neg c_2 \wedge \neg c_3)$ will be false. This will be

detected as an e-transition violation (case-1 of Theorem-1) and necessary recovery actions can be taken.

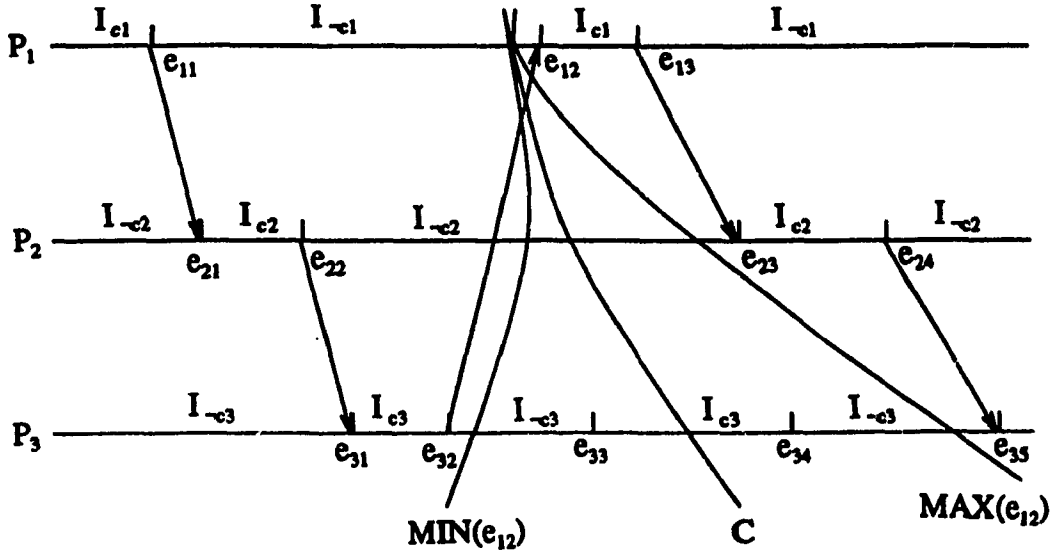


Fig. 2.2 e-transition violation due to e_{12} occurs at C

Let us say, due to design fault, Receive(TOKEN) in P_3 always signals the arrival of TOKEN. The system will violate P unless Q as illustrated above.

2.7 Conclusion

In this chapter, we identified the relationship between P unless Q and simultaneity of local predicates in a distributed system so that we could use an efficient strategy for simultaneity checking for detection of safety violations in a distributed computation. This

effectively avoids the state space explosion caused by concurrent events in distributed systems and enables us to accomplish the task with quadratic complexity at the worst. This strategy can be applied for transient fault-tolerance and distributed debugging.

Checking of simultaneity within a time window can easily be done without enumerating all the states within the window. This is possible if we make use of the causal relationship among the intervals through the start and finish time of the intervals. This leads to a linear time algorithm in terms of number of intervals for simultaneity checking once the relevant intervals are identified.

We got efficient algorithm for detection of distributed predicates that can be expressed in terms of locally detectable predicates. For certain other categories of predicates such as (i) $a = b = c = ...$, (ii) $a > b > c > ...$, where a, b, c are state variables whose values vary monotonically, efficient detection algorithms can also be designed.

In this chapter, we discussed detection of violations of the most primitive form of safety properties. We also presented an efficient detection algorithm that can be used for detection of violations of safe state (invariant) and safe transition (unless or stable) properties. As explained, for the detection of violations of safe transition properties, the monitor requires the (fault-free) global state of the system. Hence, if the global state of the system is contaminated with fault, the detection algorithm may not be able to detect violations. Since arbitrary initial state is assumed in SS systems, violations of the safe

transition properties such as form "P unless Q" and "stable P" cannot be detected. Hence, systems specified in terms of safe transition properties cannot be made SS. In the next chapter, we consider the self-stabilizing extensions of systems in the remaining category (systems whose behaviour is specified in terms of actions).

CHAPTER 3

SYSTEM CHARACTERISATION FOR SELF-STABILIZATION

In this chapter, we characterise the class of systems for which SS extension can be done automatically. We define a set of production rules by which the behaviour of the system can be derived. We define self-stabilization and self-stabilizing extension of systems in this framework.

3.1 The Model

We present a simple yet powerful model to describe the synchronization behaviour of a system. Let A be a finite set of actions. An expression containing elements from A can be derived using the following production rules.

For a,b $\in$ A :

E ::= a | a ; b | a + b | a // b

E ::= (E) | E + E | E ; E | E // E

E' ::= E* | E ; E*

The symbols ';','+','//' and '*' represents sequence, choice, concurrence and recurrence respectively. The synchronization behaviour of a distributed system is represented by E'.

A distributed system consists of a finite set of processes communicating with one another by message passing over FIFO channels with unbounded capacity. A message from a sender is delivered to the receiver within finite delay. The processes are strongly connected. A process is a set of atomic actions (send, receive and internal) written in the form of guarded commands. A process consisting of guarded commands is of the form

[] <guard> → <action> [] <guard> → <action> [] ...

The <guard> is a boolean condition defined on local variables and/or messages received. An <action> is said to be enabled when the corresponding <guard> holds. A continuously enabled action is eventually executed. A distributed system is represented by a set of partial order of events (an event is the occurrence of an action). A single partial order of events represents a run of the system (the effect of a single execution of the system). A set of partial orders represent all possible executions (the behaviour) of the system. Different executions correspond to different choices made in the course of the run of the system.

We consider distributed systems whose synchronization behaviour is represented by an expression of the form (E)*, as explained above. Extending our analysis to other forms like E;(E)* is straight forward as it is the recurrence and choices that give rise to

an infinite number of possible executions. The action symbols appearing in the expression are considered to be distinct.

3.2 Self-stabilizing Extension

Definition : A system is called ***self-stabilizing*** iff a suffix of every execution is identical to a suffix of some legitimate execution (a possible execution in absence of fault) of the system.

An execution ρ of a system S, a *labelled poset*, is represented by $[V,\prec,A,\lambda]$, where

V is a set (of *nodes* of ρ)

$\prec$ is an ordering on V (the *ordering* of ρ)

A is a finite set (the *alphabet* of ρ)

$\lambda : V \rightarrow A$ (the *labelling* of ρ)

Let $\mathfrak{L}$ be the behaviour, set of executions, of the system S. Let B be an alphabet. The *projection* of $[V,\prec,A,\lambda]$ onto B is defined as

$[U,\prec \cap (U \times U), B, \lambda\lceil U,]$, where $U = \lambda^{-1}(B)$

Let $\mathfrak{R}$ be the set of all labelled posets over an alphabet B of a system T.

Definition : A system T, is called **self-stabilizing extension** of a system S iff $\forall\ \beta' \in \mathfrak{R}$, $\exists\ \rho' \in \mathfrak{L}$ such that suffix of β' projected onto A is identical to a suffix of ρ'.

3.3 Conclusion

We present a general strategy, in Chapter-4, for obtaining self-stabilizing extension of a system whose synchronization behavior can be derived from the production rules as discussed. For the systems whose synchronization behavior is deterministic, the strategy for obtaining self-stabilizing extension is presented in Chapter-5.

CHAPTER 4

GENERAL STRATEGY FOR SELF-STABILIZING EXTENSION

In this chapter, we present a general strategy for SS extension of systems as characterised in chapter 3. We develop the augmentation strategy for fault detection and recovery. We present an algorithm to deduce the set of constraints the causal prefix of an event must satisfy. We prove that the augmentation strategy is guaranteed to detect fault, if any. We show that the recovery strategy will guarantee that the system will eventually get rid of the effects of fault. We prove that the detection & recovery strategy will produce SS extension.

4.1 Introduction

The effects of transient faults could lead to safety and/or progress violations. The program counters may assume arbitrary values due to the effects of faults because of which certain actions may be enabled erroneously. Hence, multiple executions that interlace with one another may continue resulting in persistent safety violation. Progress violations such as deadlock may arise because of loss of (synchronization) message between processes.

Safety violation is caused by interlace of multiple executions triggered by transient fault. This may arise when an action occurs without being enabled. For example, consider a resource manager system with two client processes. The synchronization behaviour of such system can be represented by (a ; (b + c))*, where a, b and c denote the actions of acquiring resource by the manager and the clients respectively. Between successive occurrences of the action a, either action b or c must occur exactly once. Hence, in every legitimate execution of such a system, the i^{th} occurrence of action a, will necessarily be preceded by the $(i-1)^{st}$ occurrence of a along with exactly one occurrence of either b or c after the $(i-1)^{st}$ occurrence of a. In general, if there are choice operators in the behaviour of a system, there may exist more than one set of actions that could occur before an action. However, given a set of actions B that enable i^{th} occurrence of an action a in a legitimate execution, the following condition must be satisfied

"each action in B must be preceded by $(i-1)^{st}$ occurrence of a"

4.2 SS extension Strategy

Two different strategies can be adopted for self-stabilization, such as (i) SS fault detection and SS recovery, (ii) integrated (distributed) fault detection and forward recovery. Many of the hand crafted SS systems belong to category (ii). However, SS extension belongs to category (i). We use approach (i) for SS extension in this chapter. A SS extension strategy based on approach (ii) is discussed in chapter 5. Our strategy for

SS extension is to determine the set of possible causal prefixes of the occurrences of each action and augment the system to maintain it during the execution. If an illegitimate causal prefix is observed at the enabling of an action, it is detected as fault and the system is forced to recover distributively. Safety violations are guaranteed to be removed by this strategy. Transient faults causing progress violations are handled by **timeout** as studied in [Mult89]. The timeout may be triggered by real-time delay or by a lower layer protocol upon detection of loss of synchronization amongst processes. We augment the system to associate a label with each event. Each process maintains a label which indicates its perception of how many times other actions in the system has occurred together with the number of times the choices have been made (left and right). Messages carry the label of the corresponding send event. The augmentation is done by adding a precondition to each action to test the validity of the causal prefix as obtained from the labels of the events that enable it. Our strategy for SS extension is described as follows :

At the enabling of an event :

(1) Check the validity of the causal prefix of the event

(2) If the causal prefix is invalid, initiate recovery

else compute the label of the event and proceed normally

4.3 Self-stabilizing Fault Detection

The *generic detection problem* is defined ([Chan88]) as follows:

For a given program F and predicates p and q, ***p detects q*** in F means that p holds within a finite time of q holding, and if p holds, so does q. Hence, the strategy for detection of fault must ensure that

(1) if the fault has caused violation of safety and/or progress, it must be detected eventually

(2) if the strategy detects faults then violation of safety and/or progress has occurred.

As per our augmentation strategy, we have to ensure that in the augmented system, causal prefix of an event will be detected invalid iff there had been violation of safety and/or progress of the system.

4.3.1 Correctness of Detection Strategy

Given an expression (E)*, a derivation tree, called the **synchronization tree(sync-tree** in short), can be constructed as follows :

(1) Create a root node labelled with E.

(2) Repeat steps 3 through 6 until no more derivation is possible.

(3) Replace a node labelled with $E_1 + E_2$ by a node labelled with '+' with left and right child labelled with E_1 and E_2 respectively.

(4) Replace a node labelled with E_1 // E_2 by a node labelled with '//' with left and right child labelled with E_1 and E_2 respectively.

(5) Replace a node labelled with E_1 ; E_2 by a node labelled with ';' with left and right child labelled with E_1 and E_2 respectively.

(6) If a node is labelled with (E), replace the label by E.

The sync-tree has leaf nodes labelled with distinct action symbols and internal nodes labelled with symbols from the set {';','+','//'}. Let $A_E \subseteq A$ represent the set of actions in E. For example, sync-tree for E = (a + (((b+c) // (d;e)) ; (f+g))) is shown in Fig. 4.1. A_E = {a,b,c,d,e,f,g}. We use a,..., g to denote actions.

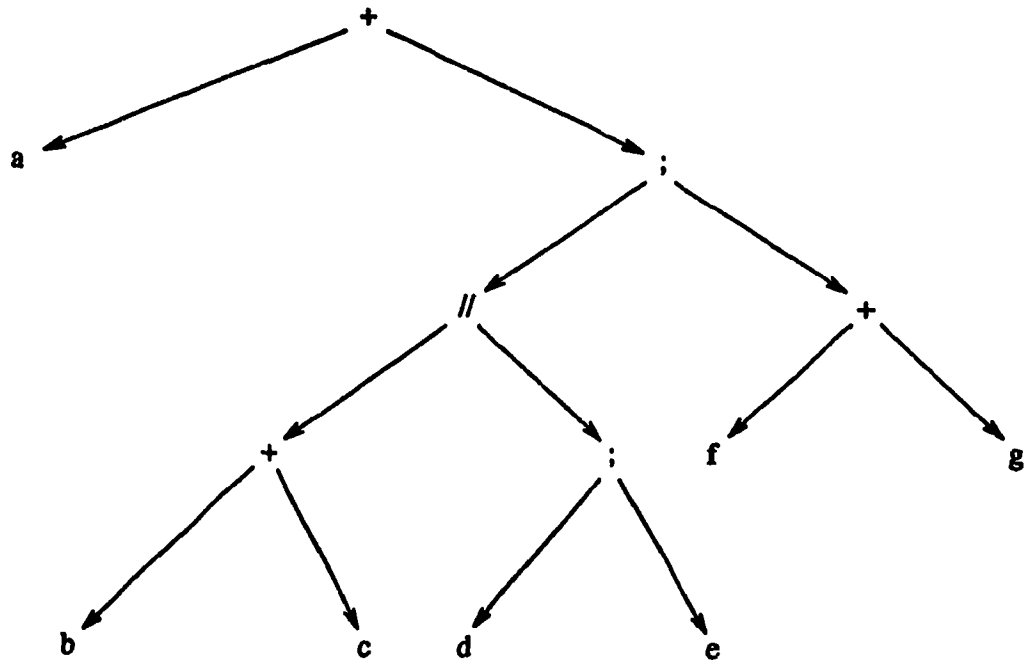


Fig. 4.1 Sync-tree of (a + (((b+c)//(d;e));(f+g)))*

Definition : A node is called **minimal common ancestor** of two leaf nodes labelled by actions b, c in the sync-tree iff b and c belong to distinct sub-trees of the former.

Let T_{bc} denote the minimal common ancestor of b and c.

An expression of the form (E)* implies that actions in E can recur indefinitely. However, in every recurrence, all the actions in E may not occur due to different non-deterministic choices made in the course of the system run. The possible subsets of actions of E that can occur in any recurrence depend upon the number of choice operators in E. Hence, an expression E can be represented by a set of **posets(partial order sets) of actions**, one corresponding to each possible combinations of choices.

Definition : Two actions b, c can co-exist in some poset of E iff the T_{bc} in the sync-tree of E is not a '+' node.

For the example, d and e can co-exist in some poset of E where as f and g cannot co-exist in any poset of E.

Definition : Two actions b, c that can co-exist in some poset α of E are ordered in α as b ; c iff the T_{bc} in the sync-tree of E is a ';' node with b appearing on the left sub-tree and c appearing on the right sub-tree T_{bc}.

For the example, b ; f can occur in some poset of E. Similarly, d ; g can occur in some poset of E.

Definition : A poset α of actions of E is a 2-tuple ($A\alpha$, $T\alpha$), where $A\alpha$ is a maximal subset of actions of E and $T\alpha$ is a partial order among actions in $A\alpha$ such that

(i) $(a,b) \in T\alpha \Rightarrow T_{ab}$ is ';' in the sync-tree of E, and

(ii) $\forall a,b \in A\alpha \Rightarrow T_{ab}$ is not a '+' node in the sync-tree of E, and

(iii) $\exists$ a poset β (= ($A\beta$, $T\beta$)) of E such that $A\alpha \subseteq A\beta$.

A poset of E can be generated from the sync-tree of E by invoking the following function with a dummy edge leading to the root as the argument.

Function **trace**(edge); { returns a poset = ($A\alpha$, $T\alpha$) }

```
Begin
  case node (the edge leads to) of
    a   : return ({a}, {∅});
    '+' : trace(left_edge) or trace(right_edge) chosen randomly;
    '//': trace(left_edge) and trace(right_edge) separately and return their union;
    ';' : trace(left_edge) followed by trace(right_edge) and return their union plus
                (∀a ∈ trace(left_edge), ∀b ∈ trace(right_edge) include (a, b) in Tα)
End;
```

Let $\mathcal{E}$ be the set of maximal posets (i.e. each poset is distinct and not a prefix of any other poset) of E. All the posets of E can be generated by traversing the left sub-tree and right sub-tree of a '+' node in separate invocation of trace. The number of maximal posets of an expression E can be determined as follows:

(1) $E = a \Rightarrow |\mathcal{E}| = 1$.

(2) $E = (E_1) \Rightarrow |\mathcal{E}| = |\mathcal{E}_1|$

(3) $E = E_1 + E_2 \Rightarrow |\mathcal{E}| = |\mathcal{E}_1| + |\mathcal{E}_2|$

(4) $E = E_1 ; E_2$ or $E = E_1 // E_2 \Rightarrow |\mathcal{E}| = |\mathcal{E}_1| * |\mathcal{E}_2|$

The posets of the example shown in Fig. 4.1 is given in Fig. 4.2.

Definition : Execution of the actions in $\alpha \in \mathcal{E}$ constitutes an **iteration** of E.

A sub-expression of the form (E_1+E_2) in $(E)^*$ implies that in any iteration of E, no action in E_2 can happen if some action in E_1 has happened and vice versa i.e the action(s) in E_1 and E_2 are mutually exclusive. A sub-expression of the form $(E_1;E_2)$ in $(E)^*$ implies that in any iteration of E, action(s) allowed to happen in E_1 must happen before any action in E_2 can happen. Hence, there is an ordering imposed between the actions happening in E_1 to those in E_2 in any iteration of E. Similarly a sub-expression of the form $(E_1//E_2)$ in $(E)^*$ implies that in any iteration of E, action(s) allowed to happen in E_1 can happen concurrently with those in E_2 and vice versa. There is no ordering

imposed between the actions happening in E_1 to those in E_2 in any iteration.

Henceforth, we use $a \in \alpha$ synonymous with $a \in A\alpha$ and $a \rightarrow b \in \alpha$ synonymous with $(a, b) \in T\alpha$. Let ${}^\circ\alpha$ (α°) represent the minimal (maximal) elements of a poset α. (i.e. ${}^\circ\alpha = \{ e \mid \nexists\, e' \rightarrow e \in \alpha \}$, $\alpha^\circ = \{ e \mid \nexists\, e \rightarrow e' \in \alpha \}$).

Definition : An action $e \in {}^\circ\mathcal{E}$ iff $\exists \alpha \in \mathcal{E}$ such that $e \in {}^\circ\alpha$.

Lemma - 4.1 : An action $e \in {}^\circ\mathcal{E}$ iff e does not belong to the right sub-tree of a ';' node in the sync-tree of E.

Proof : Direct from the semantics of ';'. ■

Lemma - 4.2 : $\forall b \in A_E$, $\exists\, \alpha \in \mathcal{E}$ and $a \rightarrow b \in \alpha$ iff $\nexists \alpha' \in \mathcal{E}$, $b \in {}^\circ\alpha'$.

Proof : ($\Rightarrow$) Say $\exists \alpha \in \mathcal{E}$ such that $a \rightarrow b \in \alpha$ but $\exists \alpha' \in \mathcal{E}$ and $b \in {}^\circ\alpha'$. $b \in {}^\circ\alpha' \Rightarrow$ b does not belong to the right sub-tree of a ';' node in the sync-tree of E (Lemma-1). This implies that $a \rightarrow b$ cannot occur in any poset of E and hence the contradiction.
($\Leftarrow$) $b \in A_E \wedge \nexists \alpha' \in \mathcal{E}$, $b \in {}^\circ\alpha' \Rightarrow \exists \alpha \in \mathcal{E}$, $a \in \alpha$ such that $a \rightarrow b \in \alpha$.■

Definition : An action a is called a **choice action** of E iff there exists a sub-expression E_1+E_2 of E such that a appears either in E_1 or E_2. Otherwise a is called a **non-choice action.**

The non-choice actions of E appear in all the posets of E. However, the choice actions of E may appear only in some posets of E. Let the choice nodes in the sync-tree be labelled by $+_1,\ldots, +_m$. We label the left and right edges of the choice node $+_i$ (**choice edge**) by l_i and r_i respectively. The sync-tree of Fig. 4.1 after the choice nodes and edges are labelled is shown in Fig. 4.3. Each choice action is associated with a unique set of choices that must be chosen for the action to be selected for execution in an iteration. Let CS(b) be the choice set of action b. CS(b) consists of the choice edges along the path from root to the node labelled b in the sync-tree. In other words, CS(b) represents the set of choice edges to be chosen at each choice node to select b for execution. For the expression as given in the example, CS(b) = $\{r_1, l_3\}$ and CS(g) = $\{r_1, r_2\}$. Each α_i is associated with a unique set of choices. Let CS(α_i) represent the choice set of α_i. CS(α_i) represents the choices to be made for selecting the actions of α_i.

$$CS(\alpha_i) = \bigcup CS(a) \text{ for } \forall a \in \alpha_i$$

α_1 = ({a}, {$\varnothing$}), CS(α_1) = { l_1 }

α_2 = ({b,d,e,f}, {(d e),(b f),(e f)}), CS(α_2) = { r_1, l_2, l_3 }

α_3 = ({b,d,e,g}, {(d e),(b g),(e g)}), CS(α_3) = { r_1, r_2, l_3 }

α_4 = ({c,d,e,f}, {(d e),(c f),(e f)}), CS(α_4) = { r_1, l_2, r_3 }

α_5 = ({c,d,e,g}, {(d e),(c g),(e g)}), CS(α_5) = { r_1, r_2, r_3 }

Fig. 4.2 Posets of expression of Fig. 4.1 with choice sets

Let **C** denote the set of choice edges in sync-tree of E.

Let $n = |A_E|$ (number of actions in E)

$m = |\mathbf{C}|$ (number of choice edges in the sync-tree of E)

We label each poset $\alpha \in \mathcal{S}$ by $L(\alpha) = (VC(\alpha), CC(\alpha))$, where

$VC(\alpha)$ is an n-element (bit) vector (vector clock of α)

$CC(\alpha)$ is an m-element (bit) vector (choice clock of α)

such that

$VC(\alpha)|_a$ (projecting $VC(\alpha)$ onto the bit assigned to a)

$=$ 1 if $a \in \alpha$

0 otherwise

$CC(\alpha)|_t$ (projecting $CC(\alpha)$ onto the bit assigned to the choice edge t)

$=$ 1 if $t \in CS(\alpha)$

0 otherwise

Definition : L' is called a **valid poset label** of E iff $\exists \alpha \in \mathcal{S}$ such that $L(\alpha) = L'$.

$\forall \alpha' \preceq \alpha \in \mathcal{S}$, $L(\alpha')$ can be similarly defined.

The valid poset labels of the posets as shown in Fig. 4.2 are :

a b c d e f g l_1 r_1 l_2 r_2 l_3 r_3

$L(\alpha_1)$ = ((1 0 0 0 0 0 0), (1 0 0 0 0 0))

$L(\alpha_2)$ = ((0 1 0 1 1 1 0), (0 1 1 0 1 0))

$L(\alpha_3)$ = ((0 1 0 1 1 0 1), (0 1 0 1 1 0))

$L(\alpha_4)$ = ((0 0 1 1 1 1 0), (0 1 1 0 0 1))

$L(\alpha_5)$ = ((0 0 1 1 1 0 1), (0 1 0 1 0 1))

Definition : $\forall a \in \alpha$, α_a is called the causal prefix leading to a in α iff (i) $\alpha_a \preceq \alpha$ and (ii) $a \in \alpha_a$ and (iii) $b \rightarrow a \in \alpha \Rightarrow b \in \alpha_a$. In other words, α_a contains a and those actions that precede a in α. The label of an action $a \in \alpha$ is defined by $L(a) = L(\alpha_a)$.

Definition : A given label L' of an action a is called **valid** iff $\exists \alpha_a \preceq \alpha \in \mathcal{E}$ such that $L(\alpha_a) = L'$.

For example, L(f) = ((0 1 0 1 1 1 0), (0 1 1 0 1 0)) is a valid label. Similarly, L(e) = ((0 0 0 1 1 0 0), (0 1 0 0 0 0)) is also a valid. Where as, L(e) = ((0 1 0 0 1 0 0), (0 1 0 0 0 0)) is not a valid label of e.

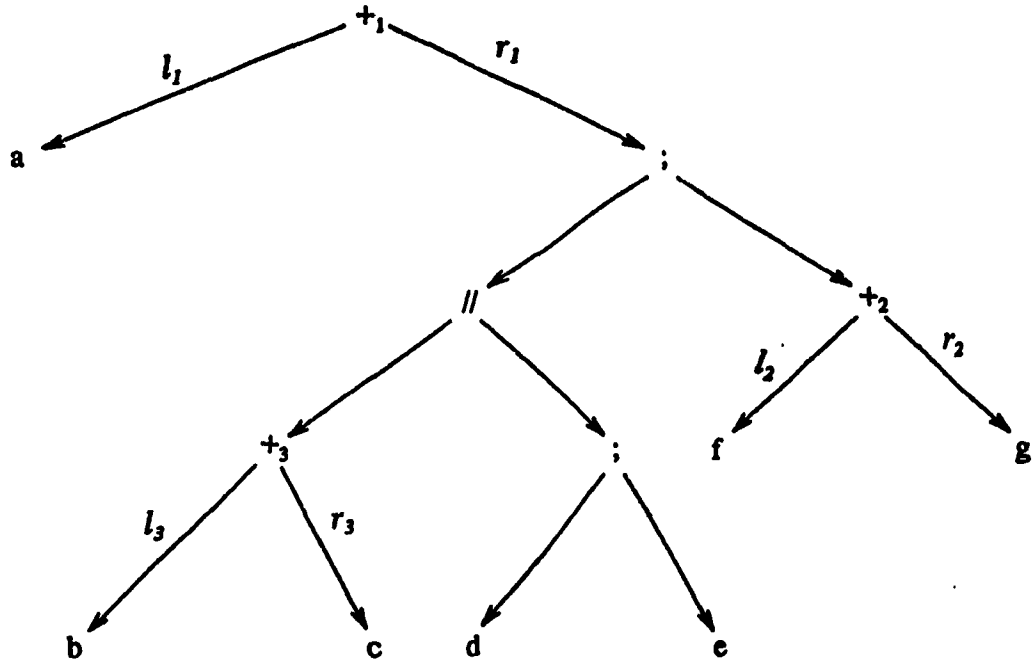


Fig. 4.3 Sync-tree Fig. 4.1 with the choice edges labelled

Definition : Two actions b and c can occur concurrently in some poset of E iff the T_{bc} in the sync-tree of E is a "//" node.

In order to select the concurrent actions b and c in an iteration, the set of choices to be made to select b(c) independent of c(b) will depend upon the possible combinations of choices in the sub-tree under T_{bc} to which b(c) belongs.

Definition : For a pair of concurrent actions b and c, the **choice differential set** of b from c (**ΔCS(b,c)**) is the additional set of choices to be made to select b in some iteration, independent of the choices to be made to select c.

For the example as shown in Fig. 1, ΔCS(b,d) = { l_3 }, ΔCS(c,d) = { r_3 } and ΔCS(d,c) = { ∅ }. There may be more than one choice differential set of b from c. Let the choice differential of c with respect to b is denoted by ΔCS(c,b). However, for an expression E, the choice differential sets ΔCS(c,b) and ΔCS(b,c) are fixed and can be determined easily from the sync-tree.

Lemma - 4.3 : Given concurrent actions b and c in E, (b//c) ∈ α ∈ $\mathcal{E}$ iff

L(b) = (VC(b),CC(b)) and L(c) = (VC(c),CC(c)) satisfy the following

(i) L(b) and L(c) are valid

∃y ∈ ΔCS(b,c), z ∈ ΔCS(c,b) such that

(ii) $\forall t \in y : CC(b)|_t = 1$

(iii) $\forall t \in z : CC(b)|_t = 1$

(iv) $\forall t \in \mathbf{C} - y - z : CC(b)|_t = CC(c)|_t$

Proof : ($\Rightarrow$) obvious.

($\Leftarrow$) L(b) is valid $\Rightarrow \exists \alpha' \in \mathcal{E}$ such that $L(\alpha_b') = L(b)$ and similarly L(c) is valid $\Rightarrow \exists\ \alpha'' \in \mathcal{E}$ such that $L(\alpha_c'') = L(c)$. From (i) and (iv) above, $\exists \gamma : \gamma \preceq \alpha_b'$ and $\gamma \preceq \alpha_c''$ and $CS(\gamma) = CS(b) \cap CS(c)$. From (iii), γ can be extended to α_b' by choosing the choices in $\Delta CS(b,c)$. Similarly, from (iii), γ can also be extended to α_c'' by choosing the choices in $\Delta CS(c,b)$. Hence, from (ii), (iii) and (iv), $\alpha_b' \cup \alpha_c'' \preceq \alpha \in \mathcal{E}$. ■

Definition : The **preset** of an action a in E is the set of actions that must immediately precede a in some poset of E. The preset of an action may not be unique.

Let Pre(a) denote the valid presets of an action a. For the example in Fig. 4.1, Pre(f) = {(b e), (c e) }.

If $a \in {}^{\circ}\mathcal{E}$, $Pre(a) = \{\varnothing\} \cup \{\ \alpha^{\circ} \mid \alpha \in \mathcal{E}\}$ i.e. nothing could happen before the action a happens for the first time and all the actions belonging to the maximal elements of some poset $\alpha \in \mathcal{E}$ must happen before subsequent occurrences of a in an execution. If $a \notin {}^{\circ}\mathcal{E}$, $Pre(a) = \cup_{a \in \alpha} \{\ b \mid b \rightarrow a \in \alpha\ \}$. The valid presets of an action can be easily determined.

All possible executions of the system is obtained by concatenation of the posets of E. Let $\mathcal{E}^*$ represent the set of executions (pomsets) of $(E)^*$.

$$\mathcal{E}^* = \{\ \alpha_{b1}\ ;\ \alpha_{b2}\ ;\ \ldots \mid \alpha_{bi} \in \mathcal{E}\}$$

Each member of $\mathcal{E}^*$ is maximal(complete) i.e. not a prefix of any other member of $\mathcal{E}^*$. If there is at least one choice symbol in E, the set of possible executions of the system is infinite i.e. $\mathcal{E}^*$ will be an infinite set. In the formalism, every event in an iteration must be completed before any event of the next iteration can start i.e. in any run of the system, all the events in the i^{th} iteration of E precedes each event in the $(i+1)^{st}$ iteration of E. If there are k posets of E, there will be k possible distinct next iteration successors of any iteration of E.

Lemma - 4.4 : All occurrences of an action a in $\alpha^* \in \mathcal{E}^*$ can be monotonically renamed as a1, a2, ...

Proof : An action occurs at most once in any iteration(α_{bi}) of E. Events between iterations α_{bi} and α_{bj} in $\alpha^* \in \mathcal{E}$ are ordered. Hence, there is no autoconcurrency of actions in α^*. Therefore, occurrences of an action a are totally ordered and can be renamed by associating monotonically increasing indexes to a. ■

All the actions in $\alpha^* \in \mathcal{E}^*$ can be renamed as per the Lemma-4.4. Once the actions are renamed, they can be treated as distinct events and the execution α^* can be treated as a poset of events (ai denotes i^{th} occurrence of the action a in α^*).

Definition : A partial execution is represented by a finite pomset

$$\beta = (\alpha_{b1} ; ...; \alpha_{bn}') \preceq \alpha^* \in \mathcal{E}^*.$$

The occurrences of an action a in β are totally ordered and can be renamed by a1, ..., ak (ak denotes the action a has occurred k times in β).

A finite prefix $\beta \preceq \alpha^*$ can be labelled by $L(\beta) = (VC(\beta), CC(\beta))$

where, $VC(\beta)|_a = k$ iff there are exactly k occurrences of a in β

$CC(\beta)|_t = k$ iff $t \in \mathbf{C}$ appears exactly k times in $CS(\alpha_{b1}), ..., CS(\alpha_{bn}')$

Let β_{ai} be the causal prefix leading to the event ai in β.

Definition : L' is a valid **event label** for ai iff $\exists \beta_{ai} \preceq \alpha^* \in \mathcal{E}^*$ such that $L(\beta_{ai}) = L'$.

In order to determine if a given label L(ai) is valid or not, it has to be decided if there exists $\alpha^* \in \mathcal{E}^*$ that contains ai whose label is identical to L(ai). If $|\mathcal{E}^*|$ is finite, it would be possible to decide the above. However, if there are choice operators in the expression, the set of possible executions of the system is infinite and hence the valid labels of ai may be infinite. Hence, it is impossible to enumerate all possible valid labels of an event ai. We present an alternate method to determine if a given label is valid for the occurrence of an action or not. For a given label to be valid for ai, it must satisfy certain constraints. The constraints to be satisfied by valid labels of ai can be deduced.

We derive the constraints to be satisfied by the valid labels of the occurrences of

an action. We label the edges of the sync-tree and obtain an edge-labelled sync-tree. The left and right edges of a node labelled by '$+_m$' in the sync-tree has already been labelled by l_m and r_m respectively.

(1) If the root node of the sync-tree is a ';' or '//' node, label the left and right edges by the variable v.

(2) Repeat step 3 until all edges are labelled.

(3) If the incoming edge to a ';' or '//' node is labelled by q $\in$ {v,l_i,r_j}, then label the outgoing left and right edges by q.

The edge-labelled sync-tree of Fig. 4.1 is shown in Fig. 4.4.

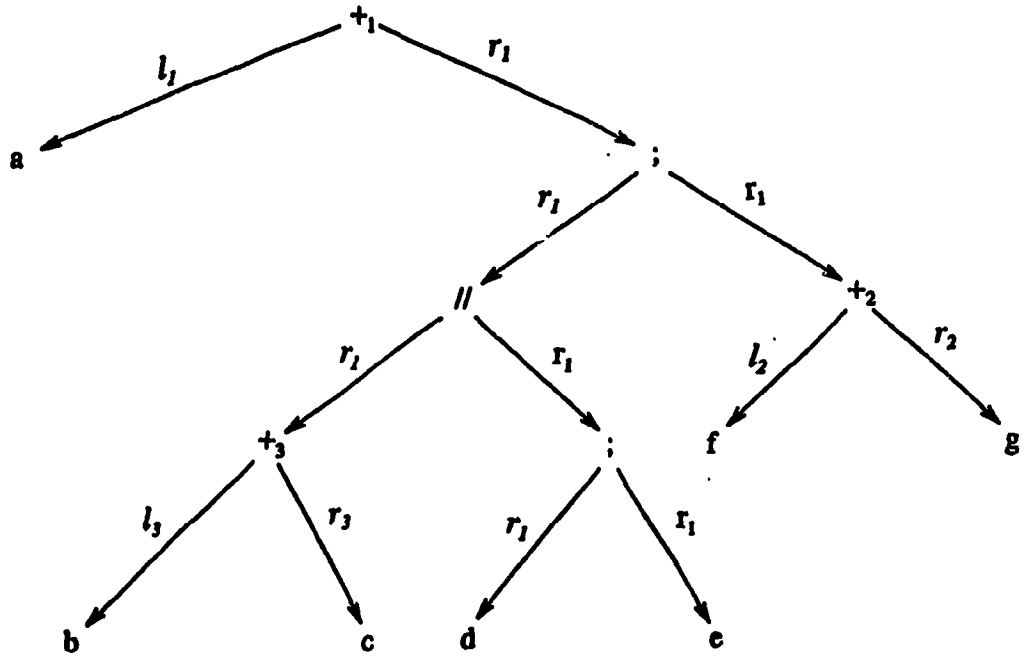


Fig. 4.4 Edge-labelled sync-tree of Fig. 4.1

In order to determine the constraints a label L' must satisfy to be valid label of dk, some of the edge-labels in the sync-tree are relabelled as follows :

Repeat

Scan the edge-labelled sync-tree on the path from root to the node labelled d

(1) If the node labelled by d belongs to the left sub-tree of a ';' or '//' node and the right edge of the latter is labelled by q, then all edges labelled by q on the right sub-tree of the latter are relabelled by (q-1).

(2) If the node labelled by d belongs to the right sub-tree of a '//' node and the left edge of the latter is labelled by q, then all edges labelled by q on the left sub-tree of the latter are relabelled by (q-1).

Until d is reached.

Let #c denote the number of times an action c occurs in valid labels of L(dk). The algorithm to deduce constraint is as follows:

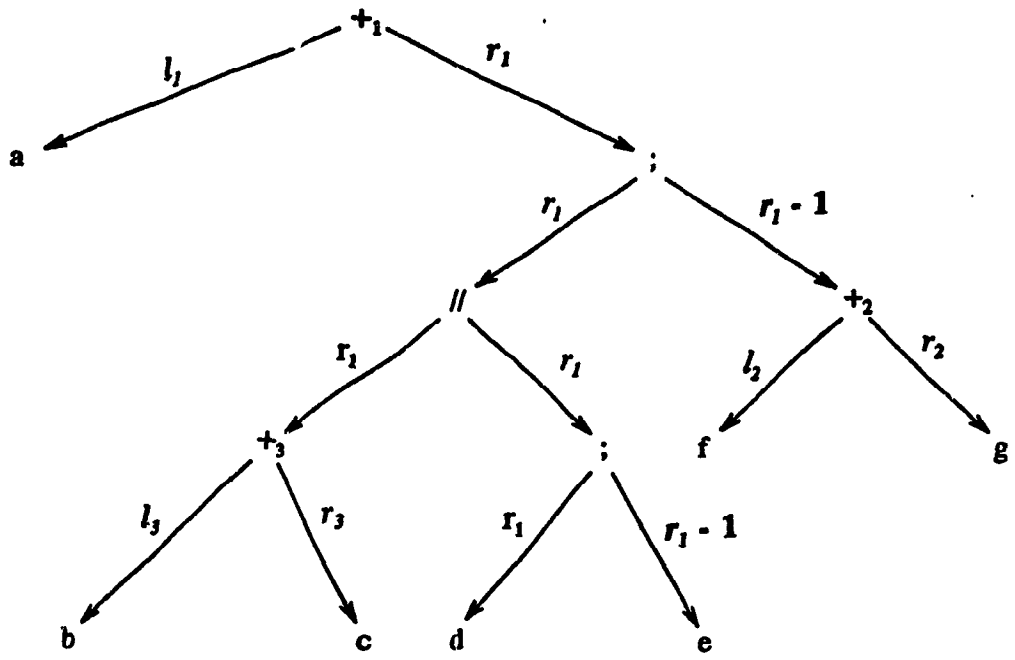
Algorithm for deriving constraints (C-ALG) :

(1) for every leaf node in the sync-tree, if an incoming edge to the node labelled by action c is labelled by q, then deduce the constraint #c = q.

(2) for all choice nodes in the sync-tree, if the incoming edge to a node labelled '$+_m$' is labelled by q, then deduce the constraint $q = l_i + r_j$

(4) Obtain the whole set of constraints at the fix-point from the set of constraints as deduced in (1) through (2).

The above procedure is guaranteed to terminate yielding a fix point because of the finite number of equations. From the description of the algorithm for deducing the constraints, it is clear that any label satisfying the set of constraints can only be a valid label of dk. In order to determine the valid labels of dk, in the expression as given in Fig. 4.1, we re-label some of the edges of the tree as shown in Fig. 4.4 to obtain the following.



4.5 Edge-labelled tree for deducing constraints to be satisfied by valid labels of dk (re-labelled edges are shown in bold face)

The constraints to be satisfied by valid labels of dk are deduced as follows : From steps (1) and (2) of C-ALG, we obtain

$\#a = l_1$, $\#b = l_3$, $\#c = r_3$, $\#d = r_1$, $\#e = r_1 - 1$,

$\#f = l_2$, $\#g = r_2$, $r_1 - 1 = l_3 + r_3$, $r_1 - 1 = l_2 + r_2$

By solving the above set of equations, the following constraints can be derived.

l_1 = #a, r_1 = #d, l_3 = #b, r_3 = #c, l_2 = #f, #e = r_1 - 1,

r_2 = #g, r_1 - 1 = l_3 + r_3, r_1 - 1 = l_2 + r_2

For example, the label ((5 5 2 8 7 3 4), (5 8 3 4 5 2)) is valid for dk (k=8). So is ((5 5 2 8 7 5 2), (5 8 5 2 5 2)), where as ((5 5 2 8 7 3 3), (5 8 3 3 5 2)) cannot be a valid label for dk (k = 8) as 8th occurrence of d in an execution, must be preceded by 7th occurrence of f and g combined. This is detected by the violation of the constraint r_1 - 1 = l_2 + r_2.

The constraints for valid labels of occurrences of all the actions can be derived using the C-ALG.

Lemma - 4.5 : A label L' is a valid label of bj iff L' satisfies the set of constraints as derived by the above procedure.

Proof : As per the semantics of execution (as derived from the trace function), among the edges of the sync-tree to be selected for execution of an action d in an iteration, can be categorised as follows : (i) that are left-edges of a ';' node (ii) that are left or right-edge of a '//' node (iii) neither (i) nor (ii) above. For (i), the corresponding right-edges are to be traced after the action d is executed. For (ii), execution of the action d does not precede the tracing of corresponding left or right edges. Hence, number of times such

edges traced preceding the execution of the action d is one less than the edges in the category (i) and (ii). ■

Lemma - 4.6 (permutation lemma):

If $\beta_a = (\alpha_{b1} ; \ldots ; \alpha_{bn} ; \alpha_a) \preceq \alpha^* \in \mathcal{E}^*$, $\beta_a' = (\alpha_{b1}';\ldots; \alpha_{bn}'; \alpha_a)$ and $(\alpha_{b1}'; \ldots ; \alpha_{bn}')$ = an arbitrary permutation of $(\alpha_{b1}; \ldots ; \alpha_{bn})$,

then $L(\beta_a) = L(\beta_a')$ and $\beta_a' \preceq \alpha^{*'} \in \mathcal{E}^*$.

Proof : Follows trivially from the derivation of executions and the definition of labels.■

Lemma - 4.7 (Concatenation lemma) :

If $\beta_a = (\alpha_{b1}; \ldots; \alpha_{bn}; \alpha_a) \preceq \alpha^* \in \mathcal{E}^*$, $\beta' = (\alpha_{b1}';\ldots; \alpha_{bm}')$ and $L(\alpha_{b1}; \ldots; \alpha_{bn}) = L(\beta')$,

then $\beta_a' = (\beta' ; \alpha_a) \preceq \alpha^{*'} \in \mathcal{E}^*$ and $L(\beta'; \alpha_a) = L(\beta_a)$.

Proof : $L(\beta') = L(\alpha_{b1}; \ldots; \alpha_{bn})$ and $(\alpha_{b1}; \ldots; \alpha_{bn} ; \alpha_a) \preceq \alpha^* \in \mathcal{E}^* \Rightarrow L(\beta')$ is valid. $L(\beta')$ is valid $\Rightarrow (\beta' ; \alpha_a) \preceq \alpha^{*'} \in \mathcal{E}^*$. Moreover, $L(\beta') = L(\alpha_{b1}; \ldots; \alpha_{bn}) \Rightarrow L(\beta' ; \alpha_a) = L(\beta_a)$.■

However, given concurrent actions b and c, and valid labels L(bj) and L(ck) of events bj and ck respectively, bj and ck may belong to two different executions. We establish necessary and sufficient conditions that must be satisfied by L(bj) and L(ck) for bj and ck to belong to the same execution.

Definition : Given concurrent actions b and c, events bj and ck are **compatible** iff $\exists$ $(\alpha_{b1};...; \alpha_{bn}; \alpha_{bc}) \preceq \alpha^* \in \mathcal{E}^*$ (after renaming all actions in α^*) such that $(bj//ck) \in \alpha_{bc}$.

Lemma - 4.8 (Compatibility lemma) : Given concurrent actions b and c, the events bj and ck are compatible iff their event labels L(bj) = (VC(bj), CC(bj)) and L(ck) = (VC(ck), CC(ck)) of bj and ck respectively satisfy the following conditions.

(i) L(bj) and L(ck) are valid

$\exists x \in \Delta CS(b,c)$ and $y \in \Delta CS(c,b)$ such that

(ii) $\forall t \in x : CC(bj)|_t = CC(ck)|_t + 1$

(iii) $\forall t \in y : CC(ck)|_t = CC(bj)|_t + 1$

(iv) $\forall t \in \mathbb{C} - x - y : CC(bj)|_t = CC(ck)|_t$

Proof : ($\Rightarrow$) obvious.

($\Leftarrow$) L(bj) is valid $\Rightarrow \exists (\alpha_{b1};...; \alpha_{bn}; \alpha_{bj}) \preceq \alpha^* \in \mathcal{E}^*$. L(ck) is valid $\Rightarrow \exists (\alpha_{b1}';...; \alpha_{bm}'; \alpha_{ck}) \preceq \alpha^{*'} \in \mathcal{E}^*$. By condition (iv) above $L(\alpha_{b1};... ; \alpha_{bn}) = L(\alpha_{b1}';... ; \alpha_{bm}')$. By the concatenation lemma, ck can be found in $(\alpha_{b1};... ; \alpha_{bn}; \alpha_{ck})$ and $L(\alpha_{b1};... ; \alpha_{bn}; \alpha_{ck}) = L(\alpha_{b1}';... ; \alpha_{bm}' ; \alpha_{ck})$. By the conditions (ii), (iii), (iv) and lemma - 4.3, $\exists \alpha \in \mathcal{E}$ such that $\alpha_{bj} \cup \alpha_{ck} \in \alpha$. Hence, $\exists \alpha^{*'} \in \mathcal{E}^* : (\alpha_{b1};... ; \alpha_{bn} ; \alpha_{bc}) \preceq \alpha^{*'}$. ■

According to the compatibility lemma, if labels of two concurrent events are valid and satisfy the conditions as specified, then they can be found in some execution of $\mathcal{E}^*$.

Corollary - 4.1 : Given a set of concurrent events S={ai, ..., ck}, if (i) labels of events in S are valid and (ii) labels of every pair of events in S are compatible, then they all can be found in some execution of $\mathcal{E}^*$.■

The label of an event L(ai) is derived from the labels of the events that enable it. Every time a occurs, the count for a is to be incremented by 1. The vector clock component of the label is the usual way i.e. for all other actions, the maximum of the counts in the labels of the enabling events is taken in the label of ai. However, for the components in the choice clock of ai, additional choices to be made in selecting a after the predecessor events have occurred must be reflected. If events bj and ck enable a, additional choices to be made for ai happen after bj and ck have happened is given by CS(a) - CS(b) - CS(c). Hence, L(ai) is computed as follows:

(1) $\forall e \in A_E$: $VC(ai)|_e = Max(VC(bj)|_e, VC(ck)|_e)$ if $a \neq e$

$= Max(VC(bj)|_e, VC(ck)|_e)$ if $a = e$

(2) $\forall t \in \mathbf{C}$ - { CS(a) - CS(b) - CS(c) }

$CC(ai)|_t = Max(CC(bj)|_t, CC(ck)|_t)$

(3) $\forall t \in$ { CS(a) - CS(b) - CS(c) }

$CC(ai)|_t = Max(CC(bj)|_t, CC(ck)|_t) + 1$

Let us say that the action a belongs to process P_i. Now, in order to detect interlace of multiple concurrent executions, it is to be ensured that events that enable ai must be

preceded by a(i-1). This can be done by checking consistency of the labels of the events that enable ai with the current label at P_i. We define the consistency condition as follows:

Definition : Let bj and ck are the events that immediately precede ai in some execution of $\mathcal{E}^*$. Let L = ((VC, CC) be the label at P_i at the time of enabling of ai. Let L(bj) = (VC(bj), CC(bj)) and L(ck) = (VC(ck), CC(ck)) be the labels of bj and ck respectively. L(ai) is said to be **consistent** with L(bj) and L(ck) iff

(1) $\forall e \in A_E : VC|_e = Max(VC(bj)|_e, VC(ck)|_e)$ if $a \neq e$

$= VC(bj)|_e = VC(ck)|_e$ if $a = e$

and (2) L(ai) is computed from L(bj) and L(ck) as mentioned above.

Theorem - 4.1 : Given β, a poset obtained after renaming all the actions of a pomset over A_E such that

(i) $\forall ei \in \beta$: preset(ei) $\in$ Pre(e)

(ii) $\forall ei \in \beta$: L(ei) is valid

(iii) $\forall ei \in \beta$: If preset(ei) $\neq \emptyset$ then L(ei) is consistent with the label of events in its preset

(iv) $\forall ei \in \beta$: the events in preset(ei), if any in β, are compatible

(v) events in β° are compatible.

iff $\exists \alpha^* \in \mathcal{E}^*$ such that $\beta \preceq (\alpha^* - \alpha^*_o)$ where $\alpha^*_o \preceq \alpha^*$.

Proof : (if) obvious.

(only if) We prove this by induction on the number of events in β. If number of events in β is equal to 1, then the theorem holds trivially.

Say, the theorem holds for $|\beta| = K$. We are obliged to prove that the theorem also holds for $|\beta| = K+1$. Let $\beta - \beta' = cr$.

Case-I : Say, $\beta' = \alpha_{b1}; ...; \alpha_{bn}$. If $\exists \alpha \in \mathcal{E}$ such that $c \in {}^{\circ}\alpha$, then $\exists \alpha^{*'} \in \mathcal{E}^{*} : (\beta'; cr) \preceq \alpha^{*}$. Let us assume that $\nexists\ \alpha \in \mathcal{E} : c \in {}^{\circ}\alpha$. This implies $\forall\ \alpha' \in \mathcal{E} : c \in \alpha', \exists\ b \in \alpha' : b \rightarrow c$ as per lemma - 4.2. Hence, preset(c) does not contain actions from any α°. This is a contradiction as L(cr) is valid, preset(cr) = $\alpha_{bn}{}^{\circ}$ and L(cr) is consistent with the labels of events in $\alpha_{bn}{}^{\circ}$. Hence, $\exists \alpha^{*} \in \mathcal{E}^{*} : \beta = \beta' ; cr \preceq (\alpha^{*} - \alpha^{*}{}_{o})$.

Case-2 : Say, $\beta' = \alpha_{b1}; ...; \alpha_{b(n-1)} ; \alpha_{bn}'$. If $\forall\ \alpha \in \mathcal{E} : \alpha_{bn}' \preceq \alpha$ and $c \in {}^{\circ}(\alpha - \alpha_{bn}')$, then $\exists \alpha^{*'} \in \mathcal{E}^{*} : (\beta' ; cr) \preceq \alpha^{*}$. Let us assume that $c \notin {}^{\circ}(\alpha - \alpha_{bn}')$. β will fail to be in some execution of $\mathcal{E}^{*}$ iff Preset(c) that should belong to α_{bn}' belong to $\alpha_{b(n-1)}$ instead. However, L(cr) is valid and consistent with the labels of events in its preset that belong to α_{bn}' leads to contradiction. ■

We present the augmentation strategy for fault detection next.

As per our strategy, an action <guard> → a at P_i is augmented as follows

<guard> →

[] <label at P_i is invalid> → <trigger reset>;

[] <label at P_i is valid> ∧

<labels of events that enable ai are valid and compatible and consistent>

→ a ; generate the label L(ai)

[] <labels of events that enable ai are invalid or incompatible or inconsistent>

→ <trigger reset>;

The application messages sent by P_i carry the label of the corresponding send event.

4.4 Self-stabilizing Recovery

The effect of the recovery is to guarantee that the system should get rid of effects of fault by ensuring that every execution has a suffix that is identical to suffix of some legitimate execution of the system. Further, for the recovery to be SS, it must guarantee the above irrespective of the state of the system at which recovery is initiated. We now develop the augmentation strategy to recover the system such that a suffix of the execution becomes identical to a suffix of some legitimate execution of the system. There are two possible strategies for recovery : (1) forward recovery and (2) backward recovery.

We present a general backward recovery strategy in this section.

4.4.1 Augmentation Strategy for Recovery

A process executes either in **reset** or **normal** state. A process in reset state restarts only after it learns that every other process in the system has received its reset call. Process P_i keeps its perception about the reset number of other processes in an array V_i of n elements (n - number of processes in the system), one for each process. Let V_{ij} represent P_i's knowledge of P_j's reset number. V_{ii} denote the reset number at process P_i. The current state of the process P_i is maintained in a variable S_i.

$\mathbf{V_{ii} = k \Rightarrow}$ P_i is in k^{th} reset if S_i = reset else P_i is executing normally after $(k-1)^{th}$ reset.

S_i = reset and $V_{ii} = k$ indicates that state of P_i has been reset for k^{th} time but it is waiting to restart i.e. P_i is in k^{th} reset state. S_i = normal and $V_{ii} = k$ indicates that P_i has restarted after $(k-1)^{th}$ reset i.e. P_i will next be in k^{th} reset state if forced to reset. Two kinds of messages are transmitted in the system (i) **reset(i,k)** - the reset message indicating that process P_i is in k^{th} reset state (ii) **appl(i,k)** - the application message sent by P_i after $(k-1)^{th}$ reset i.e. the next reset number of sender P_i will be k if it is forced to reset.

The reset module for P_i is as follows :

[] Receive(Reset(j,k)) $\rightarrow$ $V_{ii} := Max(V_{i1}, ..., V_{in})$;

$(S_i = reset) \wedge (V_{ii} < k) \rightarrow V_{ii} := k;\ V_{ij} := k;$

send(reset(P_i,V_{ii})) to all

$(S_i = reset) \wedge (V_{ii} = k) \rightarrow V_{ij} := k;$

$\forall m{:}1 \leq m \leq n{:}V_{ii} = V_{im} \rightarrow S_i := normal;$

$V_{ii} := V_{ii}+1;$ restart;

$(S_i = normal) \wedge (V_{ii} \leq k) \rightarrow$ reset local state;

$S_i :=$ reset;

$V_{ii} := k;\ V_{ij} := k;$

send(reset(P_i,V_{ii})) to all

[] Receive(appl(j,k)) $\rightarrow$ $V_{ii} := Max(V_{i1}, ..., V_{in})$;

$(S_i = reset) \wedge (V_{ii} < k) \rightarrow V_{ii} := k;\ V_{ij} := k;$

send(reset(P_i,V_{ii})) to all

$(S_i = normal) \wedge (V_{ii} < k) \rightarrow$ reset the state;

$V_{ii} := k;\ S_i :=$ reset;

send(reset(P_i,V_{ii})) to all

[] timeout $\rightarrow$ $V_{ii} := Max(V_{i1}, ..., V_{in})$;

$(S_i = normal) \rightarrow V_{ii} := V_{ii} + 1;\ S_i :=$ reset;

send(reset(P_i,V_{ii})) to all

$(S_i = reset) \rightarrow$ send(reset(P_i,V_{ii})) to all;

4.4.2 Correctness of the Recovery Strategy

The essence of the algorithm is to keep track of the reset numbers of other processes in the system. When a process in the reset state perceives that every other process in the system has received its reset message (which implies that state of all the processes has been reset with respect its most recent reset call), it restarts.

Now, we prove that the above algorithm will eventually reset the state of the system i.e. starting from any arbitrary initial state, it is guaranteed to reach a state where all R_i's have same value and S_i's are normal. We adopt the proof technique formulated by Gouda ([Goud91]). A sequence of stable predicates, such as Q1, Q2, ..., Qk are identified and these stable predicates are proved to form a convergence stair to the ultimate goal predicate Qk. In order to prove self-stabilization of the augmented system, it is to be proved that **True converges to Q1, Q1 converges to Q2** and so on, where "True" is a predicate that holds in any state of the system.

Definition : A stable predicate is a predicate on the state of the system such that if a state satisfies the predicate, all states reachable from the state also satisfy the predicate.

Definition : A stable predicate Q1 converges to a stable predicate Q2 if every state that satisfies Q1 must reach a state that satisfy Q2 in finite execution (i.e. finite number of actions executed by each process).

If the execution of the system starting from a state satisfying Qk is equivalent to a suffix of the legitimate execution, then self-stabilization is achieved. We define the set of stable predicates next.

Q1 : For all processes P_i, $V_{ii} \geq V_{ij}$

Q2 : Q1 $\wedge$ all initial messages have been received $\wedge$ $\forall i,j : V_{jj} \geq V_{ij}$

Q3 : Q2 $\wedge$ $V_{11} = V_{22} = ... = V_{nn} = C$, for all processes if no safety violation leading to a reset number greater than C is detected else

$V_{11} = V_{22} = ... = V_{nn} = C+1$

where, C is the largest reset number that exists among all processes and messages at the initial state.

The predicate "True" is assumed to be satisfied in any state.

Definition : A state satisfying predicate Q is called a Q-state.

Lemma - 4.9 : True converges to Q1.

Proof : As per the algorithm, V_{ii} is set to the maximum among V_{ij}'s as soon as either a message is received or timeout occurs, which will happen eventually. Further, V_{ii} is maintained to be maximum among V_{ij}'s. This implies Q1 is stable and a Q1-state is

reachable from any arbitrary initial state. ■

Let S_0 be the state reached after all initial messages are received from the system. This is guaranteed to happen eventually because of finite delivery of messages. All messages sent after S_0 will carry reset numbers less than or equal to that of the sender. We consider interleaved execution from S_0.

Lemma - 4.10 : Subsequent to S_0, V_{ij} is stable until $V_{jj} > V_{ij}$.

Proof : V_{ij} remains unchanged until P_i receives a message from P_j containing a reset number greater than V_{ij}. After all initial messages are flushed (i.e. subsequent to S_0), all messages from P_j to P_i will carry reset number $\leq V_{jj}$. ■

However, due to arbitrary initial values of V_{ij}, V_{ij} may be greater than V_{jj} i.e. P_i may have wrong perception about P_j's reset number. We next show that eventually, every process's perception about reset numbers of others will be less than or equal to the actual reset numbers.

Lemma - 4.11 : Q1 converges to Q2.

Proof : Consider a process P_j. Let d be the number of processes violating $V_{jj} \geq V_{ij}$. We show that d will be monotonically decreasing. As per lemma-4.10, subsequent to S_0, d

will be monotonically non-increasing.

Let d = k. Consider such a pair $V_{ij} > V_{jj}$. This implies that $V_{ii} > V_{jj}$ as per lemma-4.9. Hence, $V_{ii} \geq V_{ji}$. Eventually $V_{ji} \geq V_{ij}$ as P_j will perceive V_{ii} by receiving a message from P_i, which is guaranteed by timeout. This implies that $V_{jj} \geq V_{ji} \geq V_{ij}$ will hold eventually. Hence d will be less than k. As per the algorithm, Q2 is stable. ■

Eventually a state will be reached where d = 0. Let us call such a state to be S_1. At S_1, C is the maximum amongst V_{ii}'s.

Lemma - 4.12 : Starting from S_1, C is stable until $\forall i$, $V_{ii} = C$.

Proof : Let us assume to the contrary that $V_{jj} = C$ will increase while some $V_{ii} < C$. As per the algorithm, $V_{jj} = C$ can increase only if all V_{ji}'s = C. As per lemma-4.11, $V_{ii} \geq V_{ji}$ for all i,j. So V_{jj} will remain stable until all V_{ii}'s = C. ■

Let us call such a state to be S_2.

Lemma - 4.13 : Q2 converges to Q3.

Proof : As per Lemma-4.12, a state will be reached where each P_i will have $V_{ii} = C$. At such a state, if state of some P_j is reset, then every process will eventually receive the

reset message from P_j. Each process will perceive that every other process has received the reset message eventually and thereafter will increment its reset number to C+1 and proceed normally. From such a state, the reset numbers will only be increased if some process detects any safety violation. ■

Theorem - 4.2 : Any execution starting from S2 (a state satisfying Q3) will have a suffix identical to the suffix of a legitimate execution.

Proof : As per the algorithm, starting from S2, every process will restart normally. Hence, the execution starting from S2 will identical to the suffix a legitimate execution.■

Theorem - 4.3 : The augmentation strategy for fault detection and recovery will produce SS extension.

Proof : As per theorem 4.1, the augmentation strategy will eventually detect fault, if any. The recovery will be triggered after detection of fault. The recovery strategy will eventually cause the system execution to have a suffix identical to the suffix of a legitimate execution. ■

4.5 Conclusion

In this chapter, we presented a general strategy for the SS extensions of systems

whose synchronization specification is represented by (E)*. Our strategies for both fault detection and reset are distributed in nature. Because of this, the transient faults will be detected more efficiently and effects of faults can be contained. Hence, amount of faulty computation will be reduced by our strategy.

CHAPTER 5

SS EXTENSION OF DETERMINISTIC SYSTEMS

In this chapter, we present a general strategy for SS extension of systems with deterministic behaviour. We apply forward recovery technique for self-stabilization. Our strategy for SS extension includes the fault detection along with forward recovery.

5.1 Introduction

The behaviour of a system is deterministic iff there are no choice operators in the expression. Hence, there is only one possible execution of the system. The preset of each action is fixed and can be determined from the expression. Since, there is single execution of the system, forward rolling to a legitimate suffix distributively is acceptable. Whereas for non-deterministic systems, as discussed in chapter 4, it is not possible to roll forward distributively as the number of possible executions of the system is infinite. For systems with deterministic behaviour, logical clock can be used for self-stabilization.

5.2 SS Extension Strategy

Since, there are no choice operators in (E)*, all the actions in E occur in every

iteration. Hence, the number of occurrences of actions in the execution differ by at most one. For ease of explanation, consider each action of E to be associated with distinct processes. Say, the action a is associated with process P_a, action b is associated with process P_b and so on. The synchronization amongst processes is achieved by passing messages. Consider a system whose synchronization behaviour is given by (a // b) ; c))*. In this system, initially actions a and b can be executed. After the execution of a, P_a sends a synchronization message (token) to P_c. Similarly after execution of the action b, P_b sends a token to P_c. P_c executes c after receiving the tokens from P_a and P_b. After executing c, P_c sends two tokens, one to P_a and the other to P_b. Upon receiving the token form P_c, P_a (P_b) executes the action a (b) and sends a token to P_c and so on. The global state of such a system consists of tokens in transit and local variables at each process, if any. The global state of the system can be contaminated due to transient fault and may contain too many or few tokens incompatible with the local states of processes. Hence, execution starting from such a state could be different than expected. For example, spurious tokens created due to transient fault may enable actions causing safety violation. This may cause multiple executions interlaced with each other (safety violation). The loss of tokens due to transient fault may cause deadlock (progress violation). Our strategy is guaranteed to get rid of the safety and progress violations within a finite number of steps.

The safety requirement of the systems under consideration, as specified in (E)*, is represented by the allowed causal ordering of system events. We augment the system to maintain a logical clock at each process which indicates how many times the local

action has occurred. For systems with synchronization behaviour as represented by (E)*, causal ordering among events is fixed and can be monitored by the logical clocks of the events. For E = ((a//b); c), i^{th} occurrence of c is to be preceded by i^{th} occurrence of b and i^{th} occurrence of c. Where as i^{th} occurrence of a or b is to be preceded by $(i-1)^{th}$ occurrence of c. Let Ca be the logical clock maintained at P_a for counting the occurrences of a. Each token sent is stamped with the logical clock of the sender. Our strategy for SS-extension is to maintain a logical counter to keep count of the number of times the action has occurred. If an invalid count of enabling actions is observed at the enabling of an action, it is detected as fault and the system is forced to recover distributively. Safety violations are guaranteed to be removed by this strategy. Transient fault causing progress violation is handled through ***timeout*** assuming a lower level mechanism of the implementation as discussed before. Our SS extension strategy is given next. Say, the action a at P_a enabled by actions b and c. The code for P_a is given as follows :

Receive(token) from $P_b \rightarrow$ flag.b := true

[] Receive(token) from $P_c \rightarrow$ flag.c := true

[] flag.b $\wedge$ flag.c $\rightarrow$ do a ; flag.b := false; flag.c := false;

{do a means action a is performed and the tokens are sent out as required}

Given an expression E, the initial actions of E can be easily determined. For example, the initial action of E = (a ; (b // c) ; d) is a whereas a and b are the initial actions of the expression E = ((a //b) ; c). Similarly, the of set actions that enable a given

action in E can be determined easily. For example, the actions a and b enable c in E = ((a //b) ; c). Each process maintains boolean variables to indicate the receipt of tokens. If a is enabled by b and c, P_a maintains variables flag.b and flag.c to indicate receipt of tokens from P_b and P_c respectively. (flag.b holds implies a token from b has been received) If a is an initial action of E, the P_a is called the initiator of E. The augmentation of P_a is given as follows :

not(flag.b) → Receive(token,Cb);

if Cb ≥ Ca then flag.b := true ; Ca := Max(Ca, Cb)

[] not(flag.c) → Receive(token,Cc);

if Cc ≥ Ca then flag.c := true ; Ca := Max(Ca, Cc)

[] flag.b ∧ flag.c → Ca := Ca + 1;

do a;

flag.b := false ; flag.c := false ;

[] timeout (none of the actions at any process can be enabled) →

do a;

flag.b := false ; flag.c := false ;

(do a means the action a is performed and tokens stamped with Ca are sent out)

However, if a is not an initial action of E, the augmentation of P_a is given as follows :

not(flag.b) → Receive(token,Cb);

if Cb > Ca then flag.b := true ; Ca := Max(Ca, Cb)

if flag.c ∧ Cb = Ca then flag.b := true ;

[] not(flag.c) → Receive(token,Cc);

if Cc > Ca then flag.c := true ; Ca := Max(Ca, Cc)

if flag.b ∧ Cc = Ca then flag.c := true ;

[] flag.b ∧ flag.c → do a;

flag.b := false ; flag.c := false ;

[] timeout → do a;

flag.b := false ; flag.c := false ;

The informal description of the SS extension strategy is given next. A process waits to receive tokens from other processes that enable it (since each action is with a distinct process, "an action is enabled" is considered same as the corresponding process being enabled). If the action a is an initial action of E and a token carrying time-stamp smaller than Ca is received, it is discarded. Otherwise, the token is received and the corresponding flag is set. But if the action a is not initial action of E, a token carrying time-stamp smaller than or equal to Ca is discarded. The action a is enabled when both flag.b and flag.c holds. If a is an initial action, the new value of Ca is set to the maximum of the time-stamps received plus 1. Otherwise, the new value of Ca is set to the maximum of the time-stamps received. This is meant to eventually progress to resynchronize with the sender of the token. However, for the above SS-extension strategy to be applicable, the following essential property must be satisfied by the system.

Finiteness Restriction :

In every infinite prefix of the computation, one can find an infinite number of send or receive events of each process on each of its channels. This is necessary in order that self-stabilization can be guaranteed within a finite prefix of the execution. Failing this assumption may lead to a computation where two processes may execute infinite number of steps before reading from a third process, and hence self-stabilization cannot be achieved within a finite prefix. The systems whose behaviour can be expressed by (E)* satisfy this property trivially.

5.3 Correctness of the Strategy

In order to prove that the augmented system is self-stabilizing, the proof technique as discussed in chapter 4 is followed. The stable predicates for proving self-stabilization of the augmented program are given next. The state of the channel is represented by a sequence of time-stamps (associated with tokens) as viewed from the sender.

Definition : A channel is called consistent iff the sequence of time-stamps, as viewed from sender, form a non-increasing sequence with time-stamps less than or equal to that of the sender.

For a pair of actions a and b in an expression E, either of the following three

relationship can hold between them : (i) $a \rightarrow b$ (a precedes b not necessarily immediately) or (ii) $b \rightarrow a$ or (iii) $a \,//\, b$ i.e. $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$. The relationship between any pair of actions in E can be easily determined.

Q1 : All channels are consistent.

Q2 : Q1 $\wedge$ the logical clock at one of the initiators is largest.

Q3 : Q2 $\wedge$ $\forall a,b,c$: (i) $(a \rightarrow b) \wedge (b \rightarrow c)$ in E : $Cb + 1 \geq Ca \geq Cb \wedge Ca = Cc \Rightarrow Ca = Cb$

(ii) $(a \,//\, b) \wedge (b \rightarrow c)$ in E : $|Ca - Cb| \leq 1$

Lemma - 5.1 : Q1 is stable.

Proof : As per the augmented code, the logical clock at each process increases monotonically. Since, the sender sends tokens stamped with its logical clock, the time-stamps of the tokens sent will form a monotonically non-increasing sequence. Hence, a consistent channel will remain consistent by actions at the sender. Since, the other user of the system, receiver, removes tokens from the channel in sequence, the consistency of the channel is not destroyed by the actions at the receiver. ■

Lemma - 5.2 : True converges to Q1.

Proof : According to the finiteness assumption, there are finite number of spurious tokens in each channel initially. As per the augmented program code (the first two receive statements), the spurious tokens from the channels will be flushed by the receiver within finite number of steps. Since, the logical clock at the sender increases monotonically, the channels will be consistent with the sender within finite execution. ■

Once Q1 holds, the largest of the logical clocks will be found at one of the processes. Henceforth, we only consider the values of the logical clocks the processes as the time-stamp with the tokens on a channel are less than or equal to the logical clock of the respective sender.

Lemma - 5.3 : Q2 is stable.

Proof : As per the augmented program code, the logical clock is only incremented only at the initiator. The other processes set their clock to the maximum of the time-stamps received. Hence, once one of the initiators has the largest logical clock, it will remain so as the logical clock of other processes can be at most equal to largest time-stamp but not beyond it. ■

Lemma - 5.4 : Q1 converges to Q2.

Proof : Suppose none of the initiators has the largest clock. Say, some non-initiator P_c

has the largest clock. The largest clock at P_c, Cc, cannot increase because a non-initiator can only adopt a clock it has received. However, P_c will send token stamped with Cc either normally or due to timeout. The largest logical clock will be received by one of the initiators within finite number of steps through adoption of the clock by processes along the path from P_c to one of the initiators. Once, the largest logical clock is perceived by one the initiators, it will increment its clock to one greater than the received. Hence, one of the initiators will have the largest logical clocks within a finite number of steps. ■

Definition : Let P_a be one of the initiators having largest clock. A process pair (P_a, P_c) are called consistent iff

(i) $a \,//\, c \Rightarrow |Ca - Cc| \leq 1 \wedge \forall b$ such that $b \rightarrow c$ then $|Ca - Cb| \leq 1 \wedge Cb \leq Cc$.

(ii) $a \rightarrow c \Rightarrow Cc + 1 \geq Ca \geq Cc \wedge \forall b$ such that $b \rightarrow c \Rightarrow (P_a, P_b)$ is consistent and $Cb \leq Cc$.

Lemma - 5.5 : Q3 is stable.

Proof : Say (P_a, P_c) is consistent. Then a process P_b such that $b \rightarrow c$ is also consistent with P_a. As per the program code, execution of actions that are not initial actions do not increment the value of the logical clock. Hence, such actions will preserve the predicate. The logical clocks are only incremented at the initiators as per the program code. The actions that enable an initial action can have logical clock either equal to that of the initiator or less than that of the initiator. In the former case, the initiator can increment by 1 and R3 holds. In the later case, the initiator does not increment until all other logical

clocks become equal to its logical clock hence R3 is preserved. ■

Lemma - 5.6 : Q2 converges to Q3.

Proof : Let say, m be number of inconsistent process pairs. We are obliged to show in the convergence of Q2 to Q3 that m = k will change to m < k within finite number of steps. Let the logical clock at the initiator P_a has the largest value. Consider the subgraph of processes consistent with P_a. Let P_d be the closest process inconsistent with P_a i.e. immediate predecessors of P_d are consistent with P_a. The immediate predecessors of P_d will have clock = Ca ≥ Cd as per the program code. Hence, Cd will become equal to Ca and become consistent with P_a. Hence, the consistent subgraph will grow within a finite number of steps. ■

Theorem - 5.1 : The augmented system is SS-extension of the original system.

Proof : From lemma 5.6, Q3 will be satisfied within finite number of steps and will remain stable. The executions starting from a state will be a suffix of the legitimate execution.■

5.4 Conclusion

We presented a general strategy for obtaining SS extension of systems with

deterministic behaviour using logical clock. We demonstrate our strategy in obtaining SS extension of a token system and alternating bit protocol in the next chapter. Our strategy can also be applied to obtain SS extension of non-deterministic systems whose synchronization requirements is expressed in terms of global invariant as in SS rounds ([Dash93]).

CHAPTER 6

EXAMPLES

In this chapter, we demonstrate our SS extension strategy on three applications : (1) token ring, (2) alternating bit protocol and (3) resource manager. In all these examples, we demonstrate how our extension strategy can be applied to the original system to make them self-stabilizing.

6.1 SS extension of Token Ring

Self-stabilization was introduced by Dijkstra ([Dijk74]) using the token ring example. Self-stabilization on token ring has been presented by [Brow89], [Brun89]. However, none of these works provide any insight into design of SS systems. A reader will often get no impression of what went into the design to make it SS even for this so well-known and simple protocol. We present our SS extension of token ring step-by-step in a very simple and intuitive setting.

Consider an asynchronous directed ring of processes communicating by message passing. It is required that there should exist only one token (at most one process is previledged to execute in the exclusive mode in any state of the system) in the system

(safety property). Any process that wishes to obtain the token should get it in finite number of steps (progress property). We consider a strictly fair token management system in which processes obtain tokens in a circular fashion i.e. between two successive arrival of the token by any process, all the other processes in the system would have obtained the token once. In other words, the token moves in cycle from one process to another in the ring. For example, consider a ring of three processes P_0, P_1 and P_2. Let the orientation of the ring be $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_0$. Let a, b and c denote the action of using token at P_0, P_1 and P_2 respectively. The program for P_i is given as :

LOOP {P_i receives token from $P_{(i-1) \bmod 3}$ and sends token to $P_{(i+1) \bmod 3}$}

Receive(Token) $\rightarrow$ Use(Token); Send(Token);

END;

The synchronization behaviour of the token ring with three processes can be described by the expression E* = (a;b;c)*. The above system is not self-stabilizing. If the token is lost or more than one token come into existence due to transient fault, safety property of the system will be violated and hence the system will enter into an illegitimate state. Thereafter, the system remains in the illegitimate state forever, since the above system is neither capable of regenerating a lost token nor destroying additional spurious tokens. Since the expression does not contain any choice operator, the system behaviour is deterministic. Hence, we use forward recovery strategy as discussed in chapter 5. We now extend this system into a self-stabilizing system. Each process

maintains a logical counter. The program for P_0 is augmented as follows (augmentation is shown in bold face). Since each action is enabled by exactly one action, boolean variables are not required to record receipt of token. Instead the condition for enabling of an action and receipt of the token can be integrated.

Initially : **C0 := 0;**

LOOP

Receive(token,**C2**) → **if C2 ≥ C0 → C0 := C2 + 1;**

Use(token);

Send(token,**C2**)

[] Timeout (i.e. all the channels are empty and all the processes are idle)

→ send(token,C0)

END

The augmented code for P_1 is given as follows :

Initially : **C1 := 0;**

LOOP

Receive(token,**C0**) → **if C0 > C1 → C1 := C0 ;**

Use(token);

Send(token,**C1**)

[] Timeout (i.e. all the channels are empty and all the processes are idle)

→ send(token,C1)

END

The augmented code for P_2 is given as follows :

Initially : **C2 := 0;**

LOOP

Receive(token,C1) → **if C1 > C2 → C2 := C1 ;**

Use(token);

Send(token,C2)

[] **Timeout (i.e. all the channels are empty and all the processes are idle)**

→ **send(token,C2)**

END

6.2 SS extension of Alternating Bit Protocol

Self-stabilization of alternating-bit protocol (ABP) has been presented in [Afek89]. The original ABP (as its name implies) uses two sequence numbers. In the SS version of the ABP ([Afek89]), the transmitter uses an infinite aperiodic sequence of at least three numbers. In spite of the fact that in SS ABP, neither there are two number nor they alternate in the sequence generated at the transmitter, it is still called ABP by the authors. Another major drawback of their protocol is the redundant computation caused due to spurious messages in the initial state continue to persist. We present SS extension of the ABP that gradually eliminates the redundant computation due to initial spurious messages. Again, we demonstrate the simplicity and intuitive flavour of our strategy.

The ABP is designed to reliably transmit a sequence of data items from a

transmitter to the receiver over unreliable channels. In the original protocol, the transmitter starts with a sequence number equal to 0. Each message sent by the transmitter carries the sequence number of the transmitter at the time the message was sent. After sending the message, transmitter waits to receive acknowledgement from the receiver. When the transmitter receives an acknowledgement message carrying the sequence number equal to its sequence number, it complements its sequence number and sends the next message from the input. If the transmitter does not receive an acknowledgement within certain time, it resends the last message. The receiver initially starts with a sequence number equal to 1. When it receives a message, it writes that message to the output if its sequence number is different from what it received. After writing the message to the output, it modifies its sequence number by the sequence number of the message received and sends an acknowledgement message to the transmitter (acknowledgement message carries the sequence number of the receiver). As exemplified in [Afek89], under certain initial conditions, the protocol will not be able do its job. The original ABP is as follows :

Transmitter

Initially: B := 0;

Transitions :

[] Receive(ACK(D)) →

D = B → B := (B+1) mod 2;

Send(next_message,B);

[] Timeout → Send(cur_message,B)

Receiver

Initially : A := 1;

Transitions :

Receive(message,E) →

A ≠ E → A := E;

Write(message);

Send(ACK(A));

The external actions of the transmitter are given as following :

s0 : send a message with sequence number 0

s1 : send a message with sequence number 1

The external actions of the receiver are given as follows :

w0 : output a message with sequence number 0

w1 : output a message with sequence number 1

The expression describing the synchronization behaviour of ABP can be given as:

(s0;w0;s1;w1)*

Since, the behaviour of ABP is deterministic, we apply the strategy discussed in chapter 5 for obtaining SS extension. The augmented ABP is given as follows (augmentation is shown in bold face).

Transmitter

Initially: $B := 0$; $C_T := 0$;

Transitions :

[] Receive(ACK(D,C_R)) $\rightarrow$ $C_R \geq C_T \rightarrow C_T := C_R + 1$

$D = B \rightarrow B := (B+1) \bmod 2$;

Send(next_message,B,C_T);

[] Timeout $\rightarrow$ Send(cur_message,B,C_T)

Receiver

Initially : $A := 1$; $C_R = 0$;

Transitions :

[] Receive(message,E,C_T) $\rightarrow$ $C_T > C_R \rightarrow C_R := C_T$

$A \neq E \rightarrow A := E$;

Output(message);

Send(ACK(A,C_R));

[] Timeout (i.e. all channels empty and transmitter is idle)

$\rightarrow$ Send(ACK(A,C_R));

6.3 SS extension of Resource Manager

Consider a system consisting of a resource manager and two user processes. Let P0 be the manager and P1 and P2 be the users. The resource manager allocates resources to the user processes. Each process requests for the resource and waits to receive the

resource from the manager. After a process receives the resource, it uses and returns the resource to the manager within finite time. The manager chooses non-deterministically one of the processes as the next recipient of the resource. The resource can be with at most one them (manager or user) at any given time. Let a, b and c be the actions of acquiring resource by P0, P1 and P2 respectively.

The synchronization specification of such resource manager is given by

$$(E)^* = (a\ ;\ (b + c))^*$$

The program for P0 is given as

Repeat for ever

Receive(Resource) → a; Send(Resource);

The program for P1 is given as

Repeat for ever

Receive(Resource) → b; Send(Resource);

The program for P2 is given as

Repeat for ever

Receive(Resource) → c; Send(Resource);

The sync-tree of (E)* with edges labelled is given as follows.

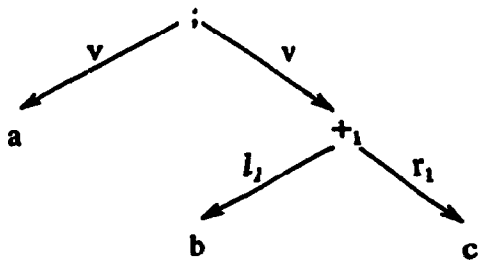


Fig. 6.3 Edge-labelled sync-tree for Resource Manager

Let the assignment of components of a label to the actions and choice edges be as follows :

$$\begin{array}{l} \quad\ a\ b\ c\ \ l_1\ \ r_1 \\ L = (\ (4\ 1\ 2)\ (1\ 2)\ ) \end{array}$$

The constraints to be satisfied by valid labels of actions can be derived as per the algorithm (C-ALG) as discussed in 4.3.1.

The constraints to be satisfied by valid labels of occurrences of a are

$$\#a = \#b + \#c + 1,\ \#l_1 = \#b,\ \#r_1 = \#c$$

The constraints to be satisfied by valid labels of occurrences of b are

$$\#a = \#b + \#c,\ \#l_1 = \#b,\ \#r_1 = \#c$$

The constraints to be satisfied by valid labels of occurrences of b are

$$\#a = \#b + \#c,\ \#l_1 = \#b,\ \#r_1 = \#c$$

The SS extension of resource manager is obtained by applying the strategy as

discussed in chapter 4, as the behaviour of the system is non-deterministic. The augmented program for P0, P1 and P2 are given as follows (augmentation shown in bold face).

The augmented program for P0 :

S_0 := normal; V_0 = (1 1 1); L_0 = (0 0 0 0 0)

Repeat for ever

[] Receive(Resource,L_j,K) → **V_{01} := Max(V_{01}, V_{02}, V_{03});**

(S_0 = normal) ∧ Valid(L_j) ∧ (V_{01} = K) ∧ (L_{01} = L_{j1}) →

a;

Update(L_0,L_j);

send(resource,L_0,V_{01})

(S_0 = reset) ∧ (V_{01} < K) → V_{01} := K;

send(reset(P0,V_{01})) to P1 and P2

(S_0 = normal) ∧ (V_{01} < K) → reset state;

V_{01} := K; S_0 := reset;

send(reset(P0,V_{01})) to P1 and P2

not(valid(L_j)) ∨ (L_{j1} ≠ L_{01}) → reset state;

V_{01} := V_{01} + 1;

send(rest(P0,V_{01})) to P1 and P2

[] Receive(Reset(Pj,K)) → V_{01} := Max(V_{01}, V_{02}, V_{03});

(S_0 = reset) ∧ (V_{01} < K) → V_{01} := K; V_{0j} := K;

send(reset(P0,V_{01})) to P1 and P2

(S_0 = reset) ∧ (V_{01} = K) → V_{0j} := K;

∀m:1 ≤ m ≤ 3:V_{01} = V_{0m} → S_0 := normal;

V_{01} := V_{01}+1; restart;

(S_0 = normal) ∧ (V_{01} ≤ K) → reset state;

S_0 := reset;

V_{01} := K; V_{0j} := K;

send(reset(P0,V_{01})) to P1 and P2

[] timeout (i.e. all the channels are empty and all the processes are idle)

→ V_{01} := Max(V_{01}, V_{02}, V_{03});

(S_0 = normal) → V_{01} := V_{01} + 1; S_0 := reset;

send(reset(P0,V_{01})) to P1 and P2

(S_0 = reset) → send(reset(P0,V_{01})) to P1 and P2;

The augmented program for P1 :

S_1 := normal; V_1 = (1 1 1); L_1 = (0 0 0 0 0)

Repeat for ever

[] Receive(Resource,L_j,K) → V_{12} := Max(V_{11}, V_{12}, V_{13});

$(S_1 = \text{normal}) \wedge \text{Valid}(L_j) \wedge (V_{11} = K) \wedge (L_{12} = L_{j2}) \rightarrow$

b;

Update(L_1,L_j);

send(resource,L_1,V_{12})

$(S_1 = \text{reset}) \wedge (V_{12} < K) \rightarrow V_{12} := K;$

send(reset(P1,V_{12})) to P0 and P2

$(S_1 = \text{normal}) \wedge (V_{12} < K) \rightarrow$ reset state;

$V_{12} := K;\ S_1 := \text{reset};$

send(reset(P1,V_{12})) to P0 and P2

$\text{not}(\text{valid}(L_j)) \vee (L_{j2} \neq L_{12}) \rightarrow$ reset state;

$V_{12} := V_{12} + 1;$

send(rest(P1,V_{12})) to P0 and P2

[] Receive(Reset(Pj,K)) $\rightarrow V_{12} := \text{Max}(V_{11}, V_{12}, V_{13});$

$(S_1 = \text{reset}) \wedge (V_{12} < K) \rightarrow V_{12} := K;\ V_{1j} := K;$

send(reset(P1,V_{12})) to P0 and P2

$(S_1 = \text{reset}) \wedge (V_{12} = K) \rightarrow V_{1j} := K;$

$\forall m{:}1 \leq m \leq 3{:}V_{12} = V_{1m} \rightarrow S_1 := \text{normal};$

$V_{12} := V_{12}+1;$ restart;

$(S_1 = \text{normal}) \wedge (V_{12} \leq K) \rightarrow$ reset state;

$S_1 := \text{reset};$

$V_{12} := K;\ V_{1j} := K;$

send(reset(P1,V_{12})) to P0 and P2

[] timeout (i.e. all the channels are empty and all the processes are idle)

→ V_{12} := Max(V_{11}, V_{12}, V_{13});

(S_1 = normal) → V_{12} := V_{12} + 1; S_1 := reset;

send(reset(P1,V_{12})) to P0 and P2

(S_1 = reset) → send(reset(P1,V_{12})) to P0 and P2;

The augmented program for P1 :

S_2 := normal; V_2 = (1 1 1); L_2 = (0 0 0 0 0)

Repeat for ever

[] Receive(Resource,L_j,K) → **V_{23} := Max(V_{21}, V_{22}, V_{23});**

(S_2 = normal) ∧ Valid(L_j) ∧ (V_{23} = K) ∧ (L_{23} = L_{j3}) →

c;

Update(L_2,L_j);

send(resource,L_2,V_{23})

(S_2 = reset) ∧ (V_{23} < K) → V_{23} := K;

send(reset(P2,V_{23})) to P0 and P1

(S_2 = normal) ∧ (V_{23} < K) → reset state;

V_{23} := K; S_2 := reset;

send(reset(P2,V_{23})) to P0 and P1

not(valid(L_j)) ∨ (L_{j3} ≠ L_{23}) → reset state;

V_{23} := V_{23} + 1;

send(rest(P2,V_{23})) to P0 and P1

[] Receive(Reset(Pj,K)) → V_{23} := Max(V_{21}, V_{22}, V_{23});

(S_2 = reset) ∧ (V_{23} < K) → V_{23} := K; V_{2j} := K;

send(reset(P2,V_{23})) to P0 and P1

(S_2 = reset) ∧ (V_{23} = K) → V_{2j} := K;

∀m:1 ≤ m ≤ 3:V_{23} = V_{2m} → S_2 := normal;

V_{23} := V_{23}+1; restart;

(S_2 = normal) ∧ (V_{23} ≤ K) → reset state;

S_2 := reset;

V_{23} := K; V_{2j} := K;

send(reset(P2,V_{23})) to P0 and P1

[] timeout (i.e. all the channels are empty and all the processes are idle)

→ V_{23} := Max(V_{21}, V_{22}, V_{23});

(S_2 = normal) → V_{23} := V_{23} + 1; S_2 := reset;

send(reset(P2,V_{23})) to P0 and P1

(S_2 = reset) → send(reset(P2,V_{23})) to P0 and P1;

Where

Update(L_i, L_j) : L_i := L_j; L_{i1} := L_{i1} + 1;

(i = 1) : L_{14} := L_{14} + 1;

(i = 2) : L_{15} := L_{15} + 1;

$$\text{Valid}(L_j) := \text{true if } (j = 0) \wedge (\#a = \#b + \#c + 1) \wedge (\#l_1 = \#b) \wedge (\#r_1 = \#c)$$

OR

$$((j = 1) \vee (j = 2)) \wedge (\#a = \#b + \#c) \wedge (\#l_1 = \#b) \wedge (\#r_1 = \#c)$$

$$:= \text{false otherwise}$$

CHAPTER 7

CONCLUSION AND FUTURE RESEARCH

Reliability is one of the most important performance requirements in application systems. Technological advancements have greatly reduced the probability of failure of components causing permanent faults. However, transient faults remain unconquered as the causes of these faults are more often related to the operational environment than to the reliability of the components of the system.

In this dissertation, we addressed fundamental issues in transient fault tolerance for the two alternative assumptions that can be made based on the effect of the faults such as (1) only application system is affected by fault and (2) both the application system and the augmentation for fault detection and recovery are affected by fault. For the former case, we considered the primitive form of safety property for detection of faults. Hence, our strategy will not only detect violations of properties specifying safe states (invariants) but also violation of properties specifying safe transitions (unless and stable). We have not come across any work on the detection of violation of properties specifying safe transitions. In this context, we claim our work as a fundamental contribution in transient fault tolerance. Moreover, we presented detection algorithms that avoid state explosion problem for a large class of useful safety properties. Our approach is useful for detection of transient faults as well as design faults (debugging applications). We concentrated on

the safety properties consisting of locally detectable predicates in this dissertation. Our future research is aimed at general predicates that are not necessarily locally detectable. We have already hinted certain forms of predicates that can still be detected without suffering from the ill effects of state explosion.

The other assumption regarding the effects of fault is considered in the domain of self-stabilization. Further, in SS systems, it is assumed that no stable store is available for data. Although a number of SS systems have been designed, all these systems stand alone in respect of design methodology. Such approaches based on custom design do not enlighten on how to make an existing system SS. The SS extension strategy as suggested in [Katz90] is one attempt in easing this difficulty. However, it is not applicable to a larger variety of systems as mentioned in the introduction. Our SS extension strategy is intuitive and based on a general methodology. We presented a general SS extension strategy based on vector clock, choice clock and version number. Our strategy consisted of augmentation for fault detection and reset. For the class of systems with deterministic behaviour, we specialized our strategy using only logical clocks. However, for systems with deterministic behaviour, forward recovery approach can be adopted for self-stabilization. We demonstrated our strategy by obtaining SS extension of token ring and alternating-bit protocol for which SS solutions have been published before. However, our SS extension demonstrates step-by-step method in obtaining such SS solution. We also obtained SS extension of resource manager, where we demonstrated our strategy to obtain SS solution of systems with non-deterministic behaviour. Our SS extension strategy can be automated through a pre-processor.

We dissected the differences of assumption, types of requirements and matched them to different strategies where efficient solutions can be developed for transient fault-tolerance. Necessity of certain assumptions to suit some strategies surfaced in this research. Our future research on SS is aimed in two important directions. (1) To extend our strategy by relaxing the restriction on behaviour of the systems. In this context, we aim to consider systems whose behaviour is expressed as a set of pomsets beyond the well-formed case as dealt with in this dissertation. (2) We want to explore other application areas for self-stabilization. The domain of applications we intend to explore for self-stabilization includes real-time systems.

BIBLIOGRAPHY

[Abad92] Abadir M. S. and Gouda M. G., The Stabilizing Computer, Proc. of the International Conference on Parallel and Distributed Systems, 1992, pp. 90-96.

[Afek89] Afek Yehuda and Brown G. M., Self-stabilization of Alternating-bit protocol, IEEE Symposium on Reliable Distributed Systems, 1989, pp. 80-83.

[Aror90] Arora A. and Gouda M. G., Distributed Reset, Proc. of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 472, Springer-Verlag, 1990.

[Aror92a] Arora A. and Gouda M. G., Closure and Convergence : A Formulation of Fault-Tolerant Computing, Proc. of the IEEE 22nd International Symposium on Fault-Tolerant Computing, 1992, pp. 396-403.

[Aror92b] Arora A., A foundation for fault-tolerance, Ph. D. dissertation, University at Texas, 1992.

[Aviz85] Avizienis A., The N-Version Approach to Fault-tolerant Software, IEEE Trans. on Software Engineering, Vol. 11, No. 12, Dec. 1985, pp. 1491-1501.

[Bril87] Brillant S. S., Knight J. C. and Leveson N. G., The Consistent Comparison Problem in N-Version Software, ACM SIGSOFT Software Engineering Notes, Vol. 12, No. 1, Jan. 1987, pp. 29-34.

[Brow87] Brown G. M., Self-stabilizing Distributed Resource Allocation, Ph. D. Dissertation, Department of Electrical and Computer Engineering, University of Texas at Austin, 1987.

[Brow89] Brown G. M., Gouda M. G. and Wu C. L., Token Systems that Self-stabilize, IEEE Trans. on Computers, Vol. 38, No. 6, June 1989, pp. 845-852.

[Burn89] Burns J. and Pachl J., Uniform Self-stabilizing Rings, ACM Trans. on Programming Languages and Systems, Vol. 11, No. 2, Apr. 1989, pp. 331-344.

[Chan85] Chandy K. M. and Lamport L., "Distributed Snapshots : Determining Global States of Distributed Systems," ACM Trans. on Computer Systems, Vol. 3, No. 1, Feb. 1985, pp. 63-75

[Chan88] Chandy K. M. and Misra J., Parallel Program Design: A Foundation, Addison Wesley, 1988.

[Chag87] Chang E. J. H., Gonnet G. H. and Rotem D., On the Cost of Self-stabilization,

Information Processing Letters, Vol. 24, No. 5, Mar. 1987, pp. 311-316.

[Char89] Charron-Bost Bernadette, Combinatories and Geometry of Consistent Cuts: Application to Concurrency Theory, Proc. of 3rd International Workshop on Distributed Algorithms, J-C Bermond and M. Raynal (editors), Springer-Verlag Lecture Notes on Computer Science 392, 1989, pp. 45-56.

[Cris85] Cristian F., A rigorous approach to fault-tolerant Programming, IEEE Trans. on Software Engineering, Vol. 11, No. 1, Jan. 1985, pp. 23-31.

[Cris91] Cristian E., Understanding Fault-tolerant Distributed Systems, Communications of the ACM, Vol. 34, No. 2, 1991, pp. 56-78.

[Dash93] Dash B. and Li H. F., Self-stabilizing algorithms for implementing Globally and Locally synchronous rounds, Proc. of the Eighth Annual University of Buffalo Graduate Conference on Computer Science, 1993, pp. 1-7.

[Dijk74] Dijkstra E. W., Self-stabilizing Systems in spite of Distributed Control, Communications of the ACM, Vol. 17, No. 11, 1974, pp. 643-644.

[Dijk82] Dijkstra E. W., EWD 391 Self-stabilization in spite of Distributed Control, Reprinted in selected Writings on Computing: A Personal Perspective,

Springer-Verlag, 1982, pp. 41-46.

[Dijk86] Dijkstra E. W., A Belated Proof of Self-stabilization, Distributed Computing, Vol. 1, Jan 1986, pp. 5-6.

[Dole90] Dolev Shlomi, Israeli Amos and Moran Shlomo, Self-stabilization of Dynamic Systems Assuming Only Read/Write Atomicity, Proc. of ACM Symposium on Principles of Distributed Computing, 1990, pp. 103-117.

[Fidg88] Fidge Colin, Timestamps in Message Passing Systems that Preserve Partial Ordering, Proc. of 11th Australian Computer Science Conference, 1988, pp. 56-66.

[Ghos91] Ghosh S., Binary Self-stabilization in Distributed Systems, Information Processing Letters, Vol. 40, No. 3, Nov. 1991, pp. 153-159.

[Goud90] Gouda M. G. and Herman Ted, Stabilizing Unison, Information Processing Letters, Vol. 35, No. 4, Aug. 1990, pp. 171-175.

[Goud91] Gouda M. G. and Multari N., Stabilizing Communication Protocols, IEEE Trans. on Computers, Vol. 40, No. 4, Apr. 1991, pp. 448-458

[Herm90] Herman Ted, Probabilistic Self-stabilization, Information Processing Letters, Vol. 35, No. 2, June 1990, pp. 63-67.

[Haba88] Haban D. and Weigel W., Global Events and Global Breakpoints in Distributed Systems, Proc. of 21st Hawaii International Conference on System Sciences, Vol. II, 1988, pp. 166-175.

[Hoov92] Hoover Debra and Poole Joseph, A distributed self-stabilizing solution to the dining philosophers problem, Information Processing Letters, Vol. 41, No. 4, Mar. 1992, pp. 209-213.

[Huan92] Huang S.-T. and Chen N.-S., A self-stabilizing algorithm for constructing breadth-first trees, Information Processing Letters, Vol. 41, No. 2, Feb. 1992, pp. 109-117.

[HsuS92] Hsu Su-Chu and Huang S.-T., A self-stabilizing algorithm for maximal matching, Information Processing Letters, Vol. 43, No. 2, Aug. 1992, pp. 77-81.

[Isra90] Israeli Amos and Jalfon Marc, Token Management Schemes and Random Walks Yield self-stabilizing Mutual Exclusion, Proc. of ACM Symposium on Principles of Distributed Computing, 1990, pp. 119-131.

[John89] Johnson B. W., Design and Analysis of Fault-tolerant Digital Systems, Addison-Wesley Publishing Company, 1989.

[Katz90] Katz Shmuel and Perry K. J., Self-Stabilizing Extensions for Message Passing Systems, Proc. of ACM Symposium on Principles of Distributed Computing, 1990, pp. 91-101.

[Kope90] Kopetz H., Kantz H., Grunsteidl G., et al, Tolerating Transient Faults in MARS, Proc. of the IEEE 20th International Symposium on Fault-Tolerant Computing, pp. 466-473.

[KooR87] Koo R. and Toueg S., Checkpointing and Rollback-Recovery for Distributed Systems, IEEE Trans on Software Engineering, Vol. SE-13, No. 1, Jan. 1987, pp. 23-31.

[Krui79] Kruijer H. S. M., Self-Stabilization (in spite of Distributed Control) in Tree-structured Systems, Information Processing Letters, Vol. 8, No. 2, Feb. 1979, pp. 91-93.

[Lamp78] Lamport L., Time, Clock and the Ordering of Events in a Distributed System, Communications of the ACM, Vol. 17, No. 8, August 1978, pp. 558-565.

[Lamp79] Lamport L., A new approach to proving the correctness of multiprocess programs, ACM Trans. on Programming Languages and Systems, Vol. 1, No.1, July 1979, pp. 84-97.

[Lamp84] Lamport L., Solved Problems, Unsolved Problems and Non-problems in Concurrency, Invited Address, Proc. of the Third ACM Symposium on Principles of Distributed Computing, 1984, pp. 1-11.

[Lamp86a] Lamport L., On Interprocess Communication: Part I - Basic formalism, Distributed Computing, Vol. 1, No. 2, Apr. 1986, pp. 77-85.

[Lamp86b] Lamport L., On Interprocess Communication: Part II - Algorithms, Distributed Computing, Vol. 1, No. 2, Apr. 1986, pp. 86-101.

[Lamp86c] Lamport L., The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication, Journal of ACM, Vol. 33, No. 2, Apr. 1986, pp. 313-326.

[Lamp86d] Lamport L., The Mutual Exclusion Problem: Part II - Statements and solutions, Journal of ACM, Vol. 33, No. 2, Apr. 1986, pp. 313-326.

[Lebl87] Leblanc T. J. and Meller-Crummey J. M., Debugging Parallel Programs with Instant Replay, IEEE Trans. on Computers, Vol. C-36, No. 4, Apr. 1987, pp.

471-482.

[LeeP90] Lee P. A. and Anderson T., Fault Tolerance : Principles and Practice, Second edition, Springer-Verlag, 1990.

[LiHF92] Li H. F. and Dash B., Detection of Safety Violations in Distributed Systems, Proc. of the International Conference on Parallel and Distributed Systems, 1992, pp. 275-282.

[LiHF93a] Li H. F. and Dash B., Self-stabilizing extensions of systems with regular behaviour, to appear in the Ninth International Conference on Systems Engineering, 1993.

[LiHF93b] Li H. F. and Dash B., Self-stabilizing Extensions of Systems with Deterministic Behaviour - A General Strategy, under preparation.

[LiHF93c] Li H. F. and Dash B., Stabilizing Nondeterministic Synchronization Behaviours, under preparation.

[LiHF93d] Li H. F. and Dash B., Safety Conformance Detection in Distributed Systems, under preparation.

[LinC92] Lin C. and Simon J., Observing Self-stabilization, Proc. of ACM Symposium on Principles of Distributed Computing, 1992, pp. 113-123.

[LuMe90] Lu Meiliu, Zhang Du and Murata Tadao, Analysis of Self-stabilizing Clock Synchronization by means of Stochastic Petrinets, IEEE Trans. on Computers, Vol. 39, No. 5, May 1990, pp. 597-604.

[Matt89] Mattern Friedemann, Virtual Time and Global States of Distributed Systems, Parallel and Distributed Algorithms, M. Cosnard et al.(editors) North-Holland, 1989, pp. 215-226.

[McDo89] McDowell C. E. and Helmbold, Debugging Concurrent Programs, ACM Computing Surveys, Vol. 21, No. 4, Dec. 1989, pp. 593-622.

[Mili90] Mili A., An Introduction to Program Fault-Tolerance, Prentice-Hall, 1990.

[Mill88] Miller P. B. and Choi J. D., Breakpoints and Halting in distributed Programs, Proc. of the 8th International Conf. On Distributed Debugging, May 1988, pp. 124-129.

[Morg85] Morgan C., Global and Logical Time in Distributed Algorithms, Information Processing Letters, Vol. 20, No. 4, May 1985, pp. 189-194.

[Mult89] Multari N., Towards a Theory for Self-stabilizing Protocols, Ph. D. Dissertation, Department of Computer Sciences, University of Texas at Austin, 1989.

[Prat87] Pratt V. R., Modelling Concurrency with Partial Orders, International journal of Parallel Programming, Vol. 15, No. 1, 1987, pp. 33-71.

[Schn93] Schneider M., Self-stabilization, ACM Computing Surveys, Vol. 25, No. 1, Mar. 1993, pp. 45-67.

[Stro85] Strom R. and Yemini S., Optimistic recovery in distributed system, ACM Trans. on Computer Systems, Aug. 1985, pp. 204-226.

[Venk87] Venkatesh K., Radhakrishnan T. and Li H. F., Optimal Checkpointing and Local Recording for Domino-free Rollback Recovery, Information Processing Letters, Vol. 25, No. 5, July 1987, pp. 295-303.

[Wald91] Waldecker B. E. and Garg V. K., Unstable Predicate Detection in Distributed Program Debugging - Extended Abstract, ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991.